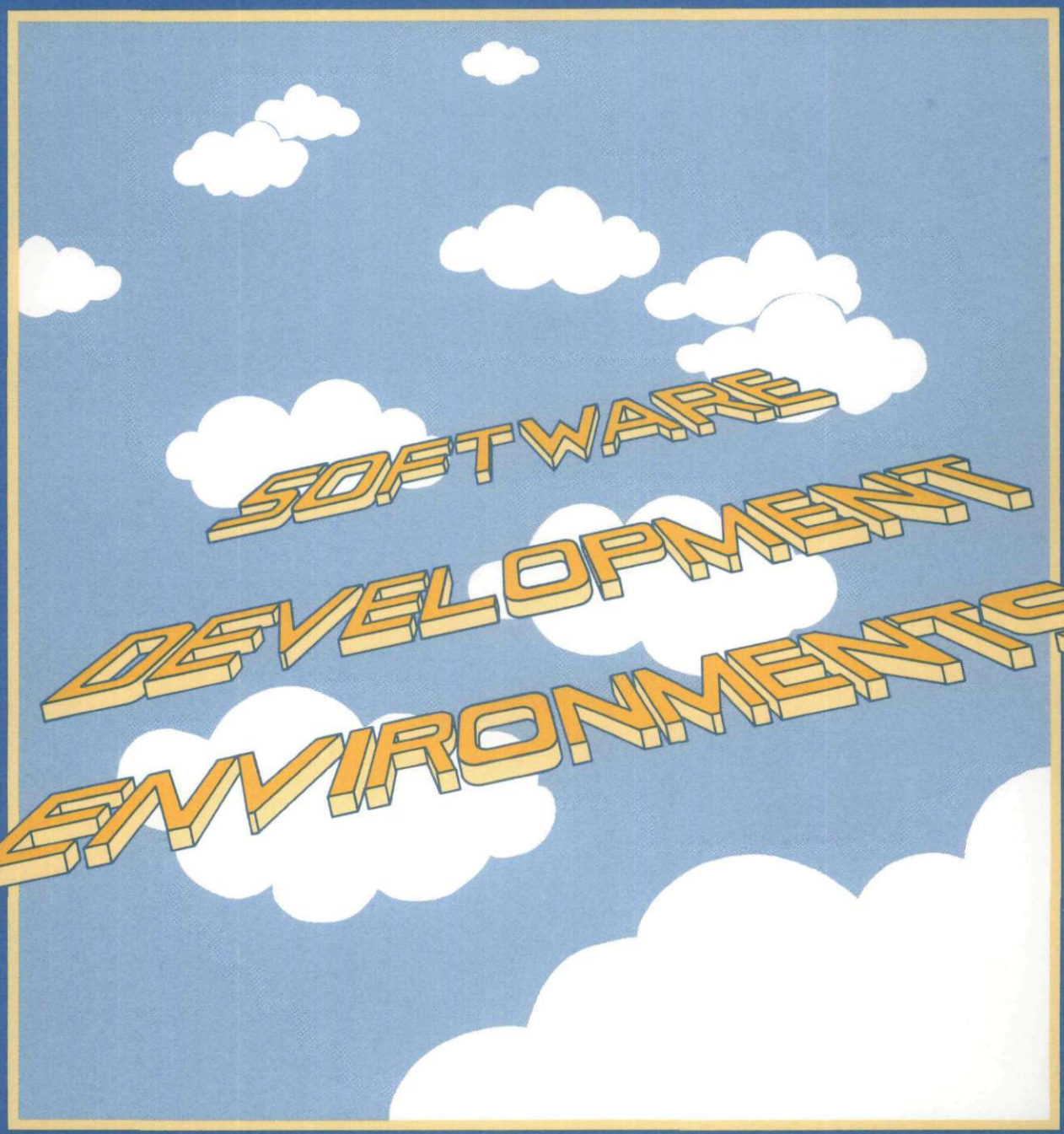
$4.00

programmer's journal

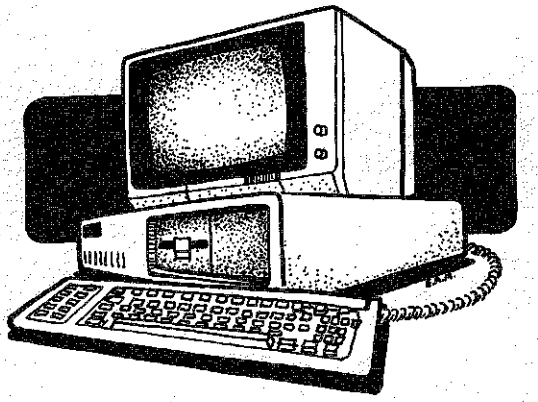
The Resource Journal for IBM PC Programmers



Volume 2 Number 3

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PROGRAMMER'S JOURNAL



CONTENTS

HOUSEKEEPING 2

LETTERS 3

PJ FORUM: REDIRECTION IN THE PC/UNIX WORLD by Frank D. Greco 6
The Unix and PC saga continues

PJ PROFILE: SOFTWARE DEVELOPMENT ENVIRONMENTS 9
The kind of hardware and software developers use

C: DATABURST, MAINFRAME-STYLE SCREEN MANAGEMENT by Steve Cabito 12
Creating a productive screen handling tool

ASM: ASSEMBLY LANGUAGE TUTORIAL/PART II by Joshua Auerbach 17
What you don't need to know to learn assembly language

BASIC: GRAPHIC WINDOWS by Ronald Moak 24
Generating graphics within windows

C: PC MUSICIAN, A COMPOSITION AND EDITING SYSTEM by Christopher Wiley 28
Create, edit and play your musical ideas

TOOLS: HYBRIDIZING ASSEMBLY AND BASIC by Wat Buchanon 32
Using the best of BASIC and the speed of assembly

dBASE II: CREATING SCREENS by Jim Bohle 35
Application screens with classy coding techniques

PASCAL: LINKING ASSEMBLER TO PASCAL AND BASIC by Corey Morrell 38
One assembly routine for both worlds

KERMIT COMMUNICATION PROGRAMS 40

MISC. 42

ISSN#0747-5861

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

HOUSEKEEPING

HOUSEKEEPING

The more programs we write, the more phantom errors we keep finding. Certainly they cannot all be our fault. The logic is correct, the code is letter perfect, and the compiler accepts it as flawless. Still, the life of a program is repeatedly bedridden with seemingly inexplicable problems. Machines can only do what they are told, so who else is telling the little buggers what to do?

Thanks to George Struble, Corey Morrell, and others who wish to remain nameless, we are finally making headway into the unexplored ether regions long unknown to software archeologist. At last, the ancient IBM System/370 Assembly runes have been mulled over and retranslated to produce THE OP CODES YOU KNEW EXISTED BUT WERE ALWAYS AFRAID TO FACE. The shackles of software programmers have been broken, the hermetic circle of the dark ones has been forced into the light. A final chapter has been added to the 370 instruction set, progenitor of all instruction sets. Read it and weep.

BRI	Branch to Random Instruction
EMIF	Erase Most Important File
IHC	Initiate Head Crash
RWD	ReWind Disk
EPI	Execute Programmer Immediate
BT	Break Tape
FMC	Fry Memory Card
JP	Jam Printer
HBWT	Hide Bug until Worst Time
BMW	Begin Malfunction if Weekend
IRE	Increase Rounding Error
SGOC	Send Garbage to Operator's Console
EHUT	Encourage Hardware Upgrade by Thrashing
DYEP	Destroy Year End Processing
IB	Insert Bug
LOET	Loop One Extra Time
RYT	Rotate Your Tires

This newly discovered instruction set is not complete, our intuition tells us there must be many more. When you find them, let us know, so we can compile a TRULY complete instruction set.

One last word: the National IBM PC Cassette Tape Archives has been established in Drain, Oregon. Forward your cassettes care of PJ so they may be added to the PC world cassette tape library. Their phone lines are always open. By the way, how many cassette tapes were OFFICIALLY released by IBM for the PC?

— Greg

□

PROGRAMMER'S JOURNAL

Publisher: Avant-Garde Creations, Inc.

Business Office:
Avant-Garde Creations, Inc.
1907 Garden Ave.
Eugene, OR 97403
(503) 345-3043

Editorial Office:
2765 Potter St.
Eugene, OR 97405
(503) 687-0398

Subscriptions and Inquiries:
P.O. Box 30160, Eugene, OR 97403
(503) 345-3043

Programmer's Journal. Volume 2, Number 3.

Copyright (C) 1984 by Avant-Garde Creations, Inc.

Published bi-monthly.

Postmaster send address changes to:
Programmer's Journal
P. O. Box 30160
Eugene, OR 97403
(503) 345-3043

Publisher: Mary Carol Smith
Editor: Greg Estes
Associate Managing Editor: Louise Shimmel
Production: Christine Baldwin

Contributing Writers:

Christopher Wiley	Ron Moak
Jim Bohle	Frank Greco
Wat Buchanon	Steve Cabito
Corey Morrell	Joshua Auerbach

Subscriptions: Bi-monthly rates are $24 U.S.
($30 in Canada, and $40 elsewhere.)

IBM and Personal Computer are trademarks of
International Business Machines, Armonk, New York
dBASE II is a trademark of Ashton-Tate.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

LETTERS

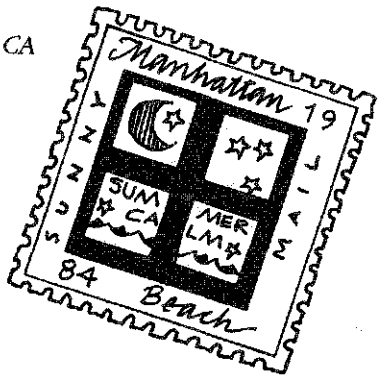
DTACK FABLE

Just about the time I had decided that the banks in Oregon were no longer honoring subscription renewal checks drawn upon Southern California banks, Vol. 2 #1 of PJ arrived, and Hoo Boy was it worth the wait.

To my general praises for the best PJ yet, let me single out the excerpts from DTACK: the "Fable" is a classic, and the "Cry Foul" should be required reading by all the MIS types who follow ComputerWorld's every procurment. At last, soothing words to a soul who was becoming paranoid because he resists converting to an OS (UNIX) that is just west of MVS in complexity. Will PJ continue to excerpt DTACK on occasion, or must I open yet another subscription? My mailman is voting for more excerpts.

The interview with Bob Wallace caught my attention, as I and several of my colleagues have been unable to alter Pascal's Heap allocation with reliability. Bob makes this sound tantalizingly easy to do, but comparison of the code on page 24 with ENTX6L leaves me no closer to my goal of making the heap smaller than 64K on large machines. Why would this be useful? To invoke several levels of Pascal programs, like a shell that invokes a major processing function which may invoke another, etc. Trouble with a 64K Heap is that memory gets gobbled up in giant bites. Would the PJ audience be interested in a tutorial on Heap control? Would Bob be willing to write it? I hope so.

Chet Floyd
Manhattan Beach, CA



WARNIER UPDATE

Vince Heiker (Letter from PJ Vol. 2 #1) took extreme exception to a remark I made in my article on structured programming (PJ Vol. 1 #6). In the article I talked briefly about Warnier diagrams and how they can be very useful. Then I said that they don't work all that well in practice. Well, I can certainly understand reader Heiker raising hell about that. They certainly DO work well in practice for MANY firms, and they've been responsible for some serious productivity gains. In fact, Editor Estes and I discussed that very point on the phone prior to publication of the article. I apologize to your readers for this totally erroneous statement.

I had hoped that your readers would detect my tongue firmly in cheek when I referred to it as a "thoroughly despicable" technique and "obnoxious" and "ugly and useless." Clearly, my reason in discussing Warnier diagrams at all was not to denigrate the practice, but to extoll it, albeit in a somewhat cynical fashion. The unfortunate statement about not working that well in practice was neither humorous nor cynical—it was flat out wrong! No wonder the true believers went berserk! I was trying to make the point that not everyone in the world likes the technique, and many people have rejected it. Those who use it, however, think it is fantastic.

By the way, I stated explicitly that the article was NOT meant as a tutorial on Warnier diagrams. Hey, how about a full article on this from Mr. Heiker? (I want a chance to complain bitterly about HIS errors and say nasty things about how astounding HIS ignorance is.) I think the subject should be covered in depth by a true believer. The fact that I said anything about the subject at all was like an anti-vivisectionist writing a how-to manual on using live dogs for rifle wound research—it's not my thing. I just wanted to expand on Joe Rayhawk's article—you guys take it from here.

In regard to the other letter of complaint, I liked reader Warner's decrementing loop solution to the quiz problem, but his statement that it would "correct the misunderstanding that Barkley left in the reader's mind" is off-base. The fact is, whether a FOR-NEXT loop performs the test first or last is trivial, as long as you know how it works on your particular machine, and don't plan to truck your software around to other machines that don't work that way. FOR-NEXT loops ought to always work one way or the other—as a standard—and I think that they don't work right on the PC, since they're really DO-UNTIL structures in that implementation. Time may very well prove that IBM is really right and the rest of the micro world was nuts all along—that's the power of marketing and short historical attention spans.

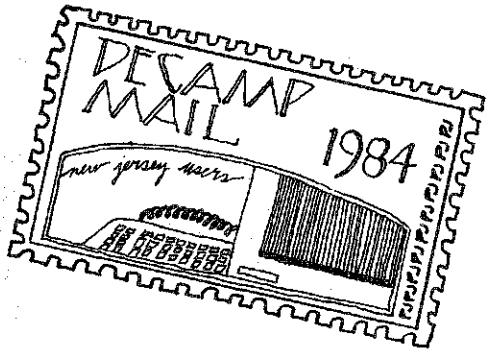
One other correction to my article: in discussing my own rule for subroutine length, I should have pointed out that in many cases I was referring to what other people would call program modules— segments of code called from a menu structure. Thus the entire procedure to add a record to a file, for example, might comprise one item from a menu and hopefully be only a page in length. A lot of custom work I do is for clients that have absurdly long record lengths with lots of fields that require error checking—if their records weren't so peculiar they d be using canned software. Consequently, even if a module uses a lot of subroutines itself, it occasionally spills over to a second source code page. Almost invariably

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

when this happens, the process can be "re-thought" and improved by further division.

From the bitterly hostile letters you received, it appears that no one out there except me ever writes a line of lousy code. I suspect that all this perfection is bound to make a programmer dour, humorless, and probably constipated as well. Now I know how poor old James Watt felt—I'm not saying one word about the ethical composition of any submissions!

Paul Barkley
Lansing, MI



LETTERS

UNIX FUROR

I'm interested in the furor about UNIX. As an operating system, UNIX is satisfactory, except that it is over-inflated, over-blown, and grossly out of proportion to the job at hand. An 8 meg operating system for personal computers is way beyond any approximation of reality, no matter what alleged virtues it may possess. Almost all operating systems are not designed, they follow the team approach to development. Consequently, the user pays heavy penalties for the employer's peculiar notion that programmers are supposed to generate code (and a lot of it), rather than quality work.

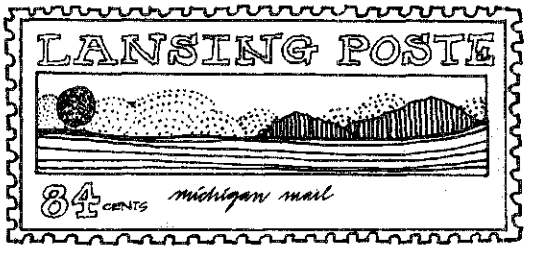
I recently completed an operating system for a military message communications processor; an interrupt-driven, multi-tasking, memory-managed embedded system which, at 1000 interrupts per second is barely breathing hard. It is 2 KB in size, with an additional memory requirement of about 6 KB for the system control tables. Admittedly, there are no device drivers, or human interface processors (these are handled by hardware) but adding those would not occupy even as much as 200 KB. I did not break any new ground with this, but I did it all by myself. I spent about six months designing it, and about one month coding and testing it. I am very parsimonious about code, and the complete opposite with documentation. The job was done in assembler (so much for ADA), but the primary path to success was the design effort, and the single handed development. If I had had "help" it would have taken a year, and been ten times the size.

Personally, I use PC-DOS, and find it adequate and uncomplicated. About the only improvements I could suggest would be to include networking, and multi-tasking, which I am given to believe will be added shortly. Eventually, when I get the time, I will design a really good system for the PC, but in the meantime, PC-DOS lets me get on with it without getting in the way. I doubt if as much could be said for UNIX.

I find your journal to be useful, and always manage to get something of value from each issue, which is all I ask.

Ralph D. DeCamp
Middletown, NJ

□

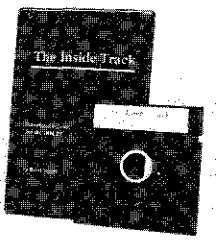


Know Thy PC!

Are you writing programs in BASIC or Pascal? The popular **Peeks 'n Pokes** has a disk with 58 programs and a 38-page manual that helps you get 'underneath the covers' of the PC. Learn how to use PEEK, POKE, INP, OUT, and DOS/BIOS function calls to do what you want, fast! Do you want to perform functions not available from BASIC or Pascal? It's all explained in the manual and demonstrated in the sample programs. Source code included!

Peeks 'n Pokes shows you how to:
- Access the system's configuration
- Unprotect BASIC programs
- Scroll part or all of the screen
- Access the file directory
- Logically swap printers
- Read and change the keyboard
- Find more Peeks and Pokes
- And much more...for only $30.00

Want to know more? **The Inside Track!** is a collection of advanced utilities for the PC programmer. It contains a disk with 61 programs, a 42-page manual, and a fold-out memory map that helps you get better performance from the PC. With this package you can give your programs assembler-assisted speed from high-level languages, get control over memory, customize and control the PC, and more. Some programs require DOS 2.00. Source code included!

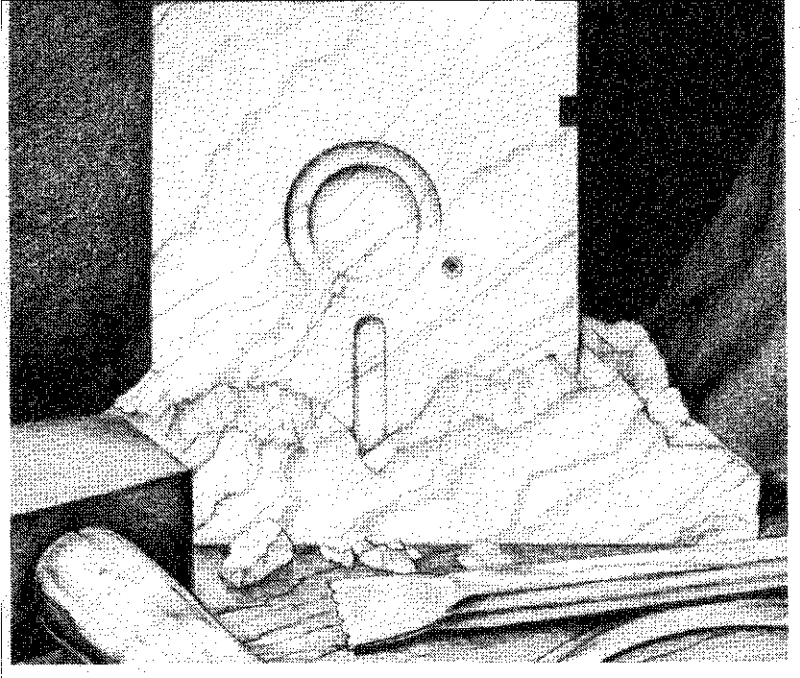


The Inside Track! shows you how to:
- Read/write files as fast as DOS
- Display data on the screen faster
- Reserve memory for your use
- Copy memory to another location
- Copy-protect your programs
- Load large programs faster
- Control the keyboard settings
- And much more...for only $45.00

MasterCard and VISA accepted. Shipping charges: $2.50 per order for UPS; $2.50 per item for First Class Mail to USA and Canada; $6.00 per item for Air Mail outside USA and Canada. Dealer inquiries invited.

Data Base Decisions • 14 Bonnie Lane • Atlanta, GA 30328 • 404/256-3860

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



"In the art of programming the difference between greatness and mediocrity is often the quality of the artist's tools."

In every top programmer's tool kit there are usually a few secret weapons. PolyMake and PolyLibrarian have become the newest essential secret weapons for hundreds of professionals who are setting new standards of productivity and performance for themselves and the industry.

Praise From Professionals

"PolyLibrarian is a powerful tool for serious hobbyists and professional programmers. It is a thing of beauty; a work of art. The documentation is excellent, the large selection of commands are intuitive to learn and easy to use, and the program itself provides a service of inestimable value. It is refreshing to see a tool of this caliber available for serious programmers. The design of this program shows foresight and ingenuity. It sets the human engineering standard for programming utilities. If all of a programmer's tools were so simple to work with, better programs could be developed in less time and with fewer headaches. *Bravo, Polytron!* I look forward to your next product."

Dan Rollins,
Programmer and Author writing in PC Age

"PolyLibrarian is an extensive, friendly utility . . . an excellent tool for serious programmers. It combines professional quality with a flexible user interface."

Greg Estes
Editor, Programmers Journal

"I am thoroughly pleased. [PolyLibrarian] is definitely one of the best products of its type that I have ever used on any system."

An Unsolicited Comment from
Steve Kauffman
Consulting Engineer

To Order or Request Literature Call

1-800-547-4000

Ask For Dept. 306

VISA & MasterCard Accepted

Or Send Checks, P.O.'s To:

Polytron Corporation DS-310-F1

P.O. Box 787, Hillsboro, OR 97123

Add $1.25 Shipping Charge For Each Product Ordered.

POLYLIBRARIAN

The Object Module Library Manager for MS-DOS™

PolyLibrarian lets you create a single library from related Objects Modules. The IBM PC Linker will then select only the modules necessary to produce an executable file. PolyLibrarian can: ☐ Create, organize, reconstruct, dissect, and compress libraries of object code modules. ☐ Add, delete and replace an object module. ☐ Extract a module from a library to create an OBJ file. ☐ Change public and external names. ☐ Optimize a module to shorten overall size of a library. ☐ Modules are time-stamped within the library. ☐ Supports full path names. ☐ Supports Microsoft's original and latest Linker (Version 2.4). ☐ Includes installation program to customize options to your specific needs. ☐ Features three Modes of Operation (Interactive, Command File, Command Line) that can be freely intermixed. ☐ Surpasses performance of other primitive librarians. ☐ Features a highly functional user interface that uses simple meaningful key words. ☐ Requires 128K RAM, MS-DOS (PC-DOS) 1.1, or 2.0. ☐ Compatible with any compiler or assembler that uses the MS-DOS (PC-DOS) Linker. ☐ Shipped with comprehensive Users Manual including step-by-step tutorial for **$99**

POLYLIBRARIAN II

Extends PolyLibrarian's power to Intel™ Format Libraries.

PolyLibrarian II is an enhanced version of PolyLibrarian. In addition to all the features and functions listed above for PolyLibrarian, this version adds a new dimension of productivity for programmers using Intel Language Products. ☐ Compatible with any Intel Format Library and any Microsoft Format Library. ☐ Compatible with Intel's Link 86 under MS-DOS. **$149**

POLYMAKE

The Intelligent Program Builder & Maintenance Tool for MS-DOS.

PolyMake frees you from the need to remember which files depend on others and which files have been modified. PolyMake will: ☐ Remember the exact sequence of operations necessary to make a new version of a program. This will significantly reduce development time, prevent bugs & manage large software projects ☐ Automatically invoke your compiler, assembler, linker, librarian, or do whatever is necessary to bring all dependent files up to date. ☐ Compare the date and time of all relevant files and use internal rules to rebuild a program or complete software system. ☐ Rules can be modified and expanded. ☐ Extensive debug capabilities help construct efficient make files. ☐ Includes features superior to UNIX Make. ☐ Fully automatic operation under MS-DOS 2.X. ☐ Full path name capabilities under MS-DOS 2.X. ☐ Requires 128K. Shipped with a comprehensive Users Manual **$99**

TM

POLYTRON

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

REDIRECTION IN THE PC/UNIX WORLD

by Frank D. Greco, AGS Computers, Inc.

Any doubts about the UNIX Operating System becoming a standard OS in the micro arena are quickly disappearing. Regardless of your stance toward this enigmatic beast, UNIX and all of its derivatives are comin' at ya.

IBM really has its eye on the burgeoning UNIX market: PC/IX on the PC, CPIX and SERIX on the Series 1, XENIX on the CS9000, and who knows what on the upcoming multi-user, multi-tasking super micro that may be in the works. At the next annual conference of commercial UNIX vendors, Uniforum, the computer giant will have the second largest booth according to Uniforum coordinator Mike Florio. Who will have the largest, you ask? AT&T, of course. Their "booth" should be the size of a two-story building, according to Florio.

Commodore is planning to announce a UNIX-based micro in the second quarter of 1984 (a CES announcement at Las Vegas, perhaps?). It is going to use the Z8000 microprocessor and run the Coherent operating system, which is a UNIX Version 7 look alike. 128K RAM and two high-density drives are going to be standard. The Mark Williams Company, author of Coherent, has said that Commodore pricing will give other UNIX micro makers a severe headache. With all the management confusion at Commodore these days, THAT remains to be seen. Commodore's other new micros, the 264 and 364, are destined for oblivion: no well-defined market niche, and, boy, are they ugly! Without Tramiel at the helm, the jury is still out with regard to Commodore's market presence.

At Apple, Steve Jobs apparently is so concerned about the up-and-coming UNIX market that he is holding several meetings with Unisoft concerning possible UNIX connections for the Macintosh (Unisoft markets UNIX for the Lisa). This should be interesting, because the MAC has no memory management which is more than strongly recommended for any UNIX-type operating system. I would really like to see how they compress UNIX into less than 128K RAM (maximum RAM available for the MAC). There have also been strong rumors that AT&T is extremely interested in licensing the Macintosh to be used in some sort of distributed computer system linked with PBXs. Remember that Apple was advertising the MAC as having a small footprint (not taking up too much desk space like a PC). Doesn't that remind you of a telephone? Interesting. Strange bedfellows, though.

Just some notes on the Macintosh: According to the Apple Developer's Handbook, the MAC contains 128K bytes of RAM, of which 21K is dedicated to the screen, 16K to the system heap, and 6K to the Pascal library. That leaves 85K RAM for developers. Of course, there's that marvelous ROM with those Quickdraw graphics routines for our use. The 400K Sony 3.5-inch drives break down to an actual 229K of usable disk capacity. 100K is needed by "system resources," 40K is used by the "Finder" (manager of application programs), 30K is dedicated to character fonts, and 1K is for boot blocks. I used a MAC at a local Apple dealership and found it to be amazing. And the new Lisas at $3500 to $4500 are all of a sudden very attractive. What is more impressive about these machines is that Microsoft has an incrementally compiled BASIC, and Think Tank Technologies has a true Pascal interpreter for the MAC. When programmers use Microsoft BASIC they can open one window to display the output, another window to display the listing and still another window to display the current executing line. As far as the Pascal interpreter goes, the education market has been waiting a long time for a true Pascal interpreter. All we need now is a C language interpreter and we're cooking. Sorry, all you PC chauvinists out there, but I think the MAC is going to prove to be a very viable alternative to the PC. Let's face it: the MAC has a 16-bit 68000 in there, and the PC has an 8-bit 8088 (all Intel publications describe it as an 8-bit).

Since we're talking chips, at a recent local users' group meeting an Intel spokesman said that the iAPX386, their 32-bit version of the 8086, will have both segmentation and linear addressing. When questioned further, he explained that system programmers will have access to several 32-bit segment registers, which implies a maximum addressability of four gigabytes. The programmer has a choice of using segments or just addressing memory linearly as one very large segment. Now, this makes more sense; segmentation is a reasonably good idea, but limiting the segments to 64K (as on the 8086/8088) is ludicrous. What happens when you are updating a high-resolution bit-mapped graphics screen that represents much more than 64K? The programmer has to keep track of how many display bytes/words are going to the memory map and, when the count reaches the 64K limit, switch to another segment. This procedure should not be necessary, at least to my way of thinking. What about real-time applications, where timing is crucial and the sudden jumping to another segment upsets the timed balance of things? Either confine yourself to 64K or use a different breed of processor.

Speaking of 8086 news, Microsoft is hard at work on a 80286 add-on processor board for the PC that will run XENIX. We have not seen it yet because Microsoft feels that the UNIX market has not been primed enough. Manufacturers have to time the market just like surfers who wait and wait for just the right wave.

Another interesting Microsoft tidbit: Microsoft has started an Artificial Intelligence Expert Systems Products group for the imminent AI wave. Obviously, with the retailing experience

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

of Microsoft's president Jon Shirley (garnered at Tandy), this new group will have a consumer orientation. Apparently, Microsoft has more than an academic interest in the Japanese fifth-generation project: it looks like the AI influence was rubbed off from their Japanese dealings. Remember their Japanese connection, Kyocera, worked with Microsoft (and of course Tandy) to develop the Model 100 lap computer. What's more, the Microsoft small microcomputer standard MSX is very popular in Japan. I am told that Japanese high school students now use MS-DOS based lap computers to practice their PROLOG (artificial intelligence language).

We can already see Shirley's influence on Microsoft by looking at the latest Microsoft WORD advertisements. Very cartoony, I'd say. Remember the Microsoft toolbox ads which conjured up serious thoughts of programmer tools? The WORD ads have a mouse rounding up some floppy disks, cowboy style. Well, expect to see more consumer-oriented ads and consumer-oriented products. I would bet money that Microsoft has a Lotus-like product in the wings along with a dBASE II style product. Right now, they are trying to attack the WordStar people with WORD.

How about their new name for MBASIC: GW BASIC. It's supposed to represent "Gee Whiz BASIC." Maybe it really means "Gates William BASIC." This industry gets more titillating every day.

□

The IBM/PC can dance—Use C86.

OPTIMIZING C86™ controls Charlie... LIKE A PUPPET ON A STRING!

Serious programmers can use Optimizing C86 to control the IBM PC and MS-DOS computers. Features include 30% + more speed and:

- A **Full** and **Standard** implementation of the C language balancing tight control of the machine with productivity and portability.
- **Hardware and operating system interfaces** for: graphics, interrupt control, **8087** use, I/O ports, real time applications and for producing ROMs.
- Use of the **standard MS-DOS Linker** and a compiler option to produce **MASM** Assembler output. Helps integrate with MASM, MS FORTRAN and PASCAL.
- **A rich set of libraries** that includes source for K&R functions, extras for string handling, graphics, sorting, floating point (8087 and 8086/8088), **"Large" model** memory use (1,000K RAM), **"Small" memory** model, MS-DOS 1.1, 1.25, 2.0, 2. + +.
- Support for numerous **add-on libraries** including: HALO **Graphics,** C Tools for fundamentals in many areas, PHACT for **ISAM,** and others. Ask for a list and for descriptions.

Pull Charlie's strings with our fast, complete, proven, reliable C Compiler—the leading compiler for serious programmers of MS-DOS and CPM-86 systems. Still only $395. Call for details.

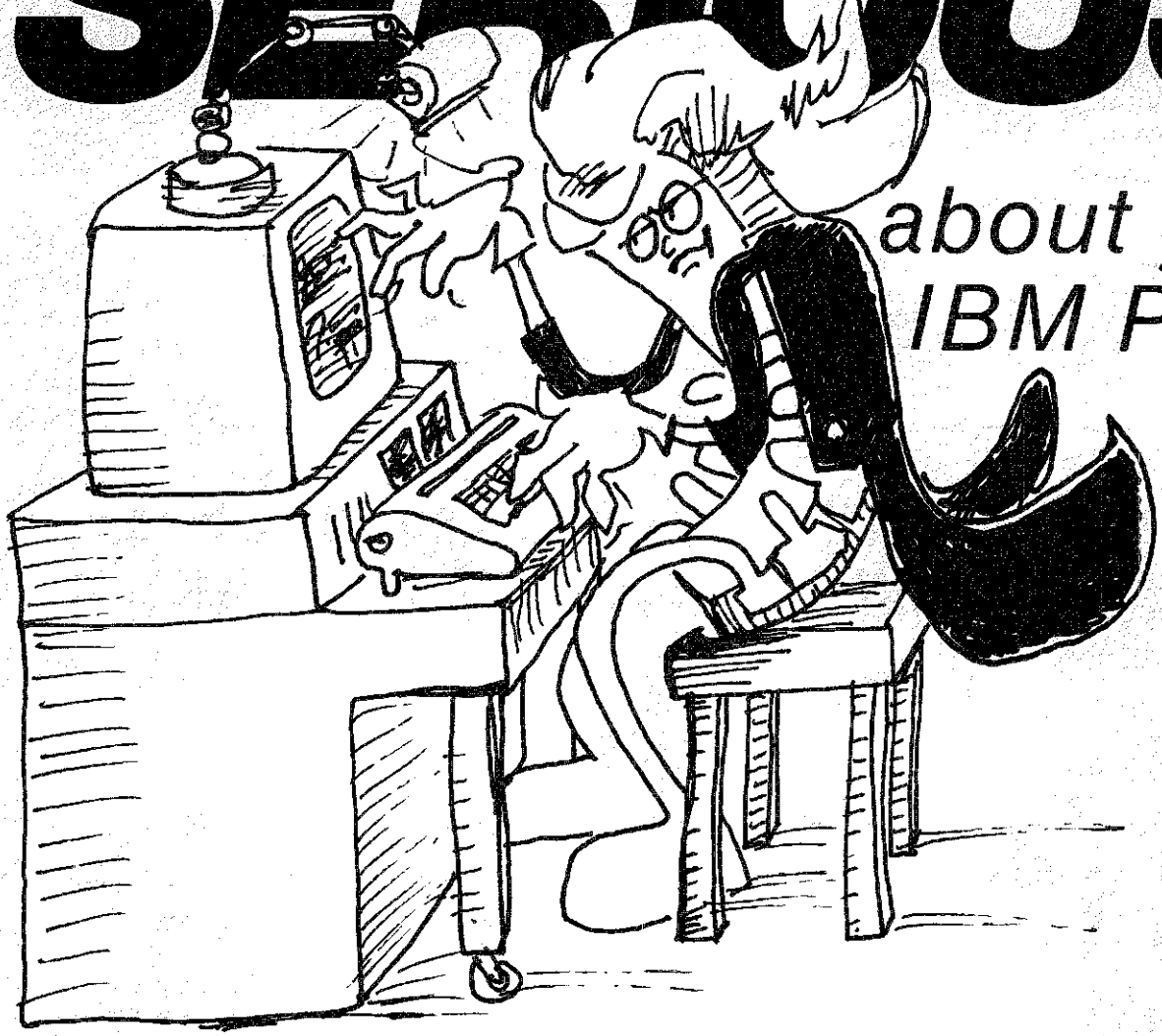
INQUIRIES AND ORDERS:
800-922-0169
Technical support: (201) 542-5920
Note: Prices subject to change without notice.

Computer Innovations
980 Shrewsbury Avenue
Suite J-950
Tinton Falls, NJ 07724
Visa MC

C86 is a trademark of Computer Innovations, Inc. CPM-86 is a registered trademark of Digital Research. MS-DOS is a trademark of Microsoft. PC-DOS is a trademark of International Business Machines.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

SERIOUS



about your IBM PC?

READ ***the Definitive Journal for the IBM Personal Computer User.*** Too serious about your IBM PC to plow through 3 lbs. of ads...industry history...revealing personal interviews? Consider Personal Computer Age. Written and published monthly by serious computerists for serious computerists, Personal Computer Age presents PC technology from fundamentals to hands-on tweaking...And explains it so you can use it.

Start your subscription now. Personal Computer Age... one year 12 informative issues...just $24.00.

The Definitive Journal for the IBM Personal Computer User
Personal Computer Age

Department 6083
P.O. Box 70725
Pasadena,
California 91107
(800) 824-7888
operator 77

Enter my subscription today!

☐ New
☐ Renewal
☐ Payment enclosed
☐ Bill me

☐ 1 yr. (12 issues) $24.00
☐ 2 yrs. (24 issues) $45.00
☐ 3 yrs. (36 issues) $64.00
☐ Send information on back issues, too!

charge my:
☐ Visa ☐ MASTERCARD Exp. Date_____

Card # _____

_____
(authorization signature)

1 yr subscription: Canada & Mexico $29.00
Other countries via Air . $90.00
Surface mail $40.00
Payment made in U.S. funds drawn on U.S. bank.

Name _____

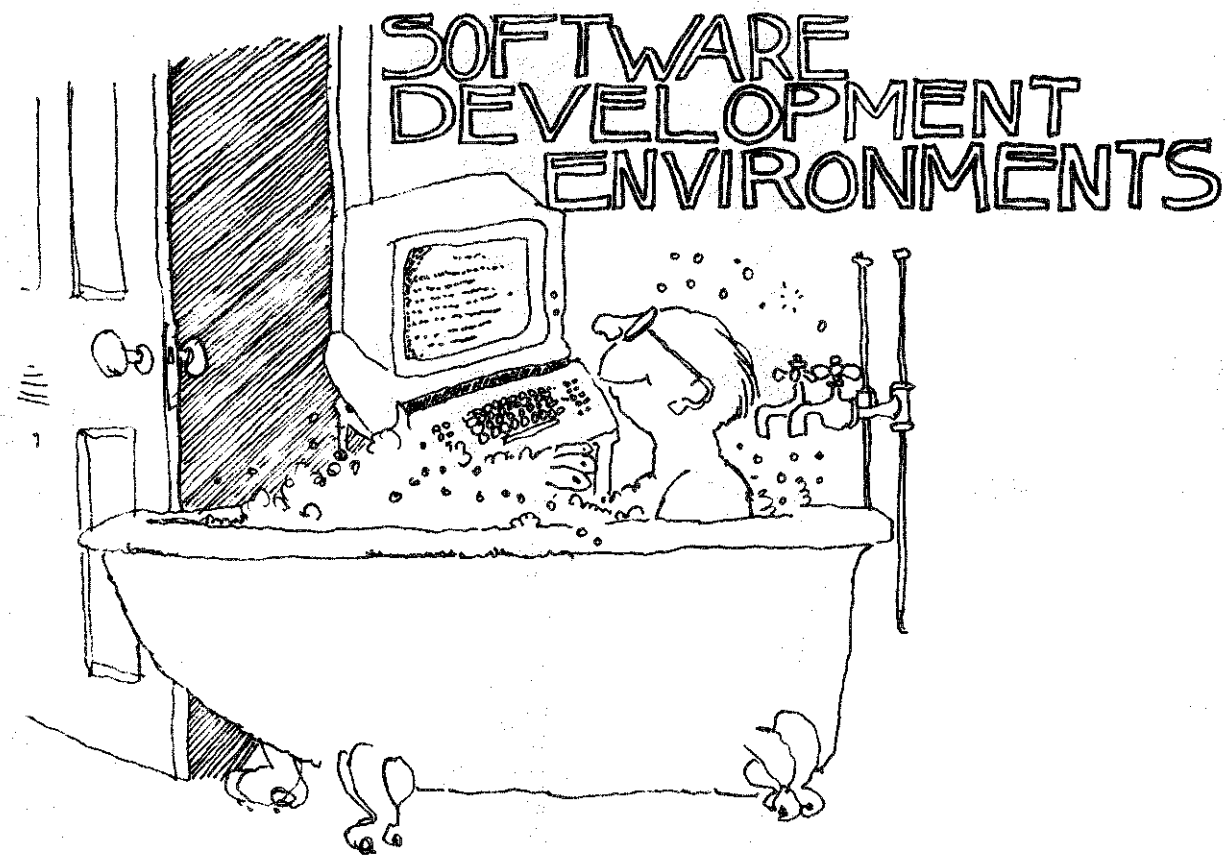
Address _____

City _____ State _____ Zip _____

The Definitive Journal for the IBM Personal Computer User
Personal Computer Age

Department 6083
P.O. Box 70725
Pasadena,
California 91107
(800) 824-7888
operator 77

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



a brief look at the personal preferences of software developers

Chet Floyd, software designer

PJ: Chet, what kind of programming do you do?

CHET: Applications, usually. Much of what I do involves the management of large amounts of data. I've done generalized database managers as well as vertical market software. Right now I am writing some data and text management software for a company.

PJ: What compilers and assemblers do you use?

CHET: We are using Microsoft's Pascal compiler. We just switched over to it; previously we were using the IBM Pascal compiler. We are very, very pleased with the upgrade that Microsoft has written.

PJ: Was it hard to convert the code?

CHET: No. The Microsoft compiler is 100 percent compatible. You would expect that, since they wrote the IBM package. There are some undocumented bugs in the IBM Pascal compiler. I first started running into this at Comdex in November. I was talking to IBM, and the people there knew little about the compiler, (I used to work for IBM years ago). Then I went over to the Microsoft booth and they gave me glazed looks concerning the problems I was having with the IBM software.

It occurred to me that IBM Pascal is basically an unsupported piece of code these days, even though some help is provided through dealers. The compiler seems like an orphan. So I decided to go with Microsoft's Pascal. The only problem I had during conversion was, first, the Microsoft compiler was catching errors that the IBM version was ignoring. It was in the area of subscripting. I would select element zero when the array started with element one. The IBM compiler didn't catch it at all.

PJ: Any other compilers?

CHET: We have Computer Innovations C86 and the Macro Assembler.

PJ: Which text editor do you use?

CHET: The Personal Editor.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PJ: What is your hardware like?

CHET: I spend most of my time on an XT. I also have a PC, it's known as LPT1: around here. The XT has an AST SixPack in it. I use their Ramdisk as well as Multi-Job, which gives me concurrent processing.

PJ: How do you set up the Ramdisk, compiler and multi-tasking software?

CHET: I have what I consider to be a pretty dynamic environment. In one partition I have the Pascal compiler, and do all of my testing there. In the other partition I keep my editor up with the source code available. Any changes that need to be made as a result of testing can be accomplished immediately. Multi-Job does not restrict files to any particular partition. Also, I use Blaise's Pascal Tools extensively; I keep their interface files on the Ramdisk. The Pascal work files go on the Ramdisk too.

PJ: Have you run across any unresolvable software problems?

CHET: None, except for redirection of I/O from an application program. Redirection seems to work fine for the DOS 2.0 utilities and Blaise's Tools, but, it turns out, you must explicitly place an EOF character in your input file.I found it such an ungodly pain to keep inserting the EOF character using EDLIN that I have avoided it entirely. This is all in reference to keyboard input.

Kurt Klinzing, author of C Building Blocks

PJ: Kurt, after looking over several of your C software tool libraries, I can see why programmers feel their money was well spent. The volume of C functions you have made available is astounding. There must be over 500 functions in just the packages we have received.

KURT: Yes, that's true, and I add to it every day.

PJ: Besides the extensive set of C utility routines, you have a database manager series and a complete Hayes modem control library.

KURT: The documentation for the modem package is being copy edited and will be ready shortly.

PJ: Besides your C libraries, which you have recently begun to market, what other kind of programming have you done?

KURT: Let's see, what projects have I completed in the last year. I did an artist's graphic work station for Time/Life books, and a money manager for them (that was not fun). I felt their functional specification was not the best, but they were paying a lot and they insisted the design was exactly what they wanted.

PJ: Do you use languages other than C?

KURT: I do have the assembler, but I have written 50,000 lines of C code and only about 200 lines of assembler code. I only use assembler if I am backed into a corner. The editor I use is PMATE.

PJ: What hardware do you use?

KURT: I have a PC with two floppies and a 20-megabyte Tall Grass with a cartridge backup, Hayes modem, and two monitors.

PJ: Do you use a Ramdisk for compiling?

KURT: I just use the hard disk for development. I have a spooler which I found via the Freeware program. The only other unique software I use is an archiving system for saving source code. Functions are stored in large files and then accessed whenever they are needed. This also compresses them somewhat. It also serves as a handy backup system. The archiving system comes with many of the libraries that I market. It creates a database file with B-trees to handle updating the source files. This makes it easy to do a directory of the source code routines.

PJ: Do you remember when you started programming in C?

KURT: Yes, it was around a year and a half ago - almost two years now. I already knew PL/1 so I was able to pick it up pretty fast.

Brett Salter, author of Peeks 'n Pokes

PJ: Brett, how are the sales of Peeks 'n Pokes and Inside Track going?

BRETT: Very well - this should be our best month ever.

PJ: I know you used to do mainframe consulting work, and are now happily self-employed as an independent software programmer and marketer for IBM PC products. What is your development system like?

BRETT: I have three systems: Papa Bear, Mama Bear, and Baby Bear. The XT has 576K with a monochrome monitor, and I do

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

most of my development on it now. The other two systems are a PC and PCjr.

PJ: What compilers do you use?

BRETT: I use Microsoft's Assembler. I am head over heels in love with it. I also have the IBM Assembler, BASIC compiler, and MS/Lattice C compiler.

PJ: You mentioned the speed of the Microsoft Assembler at the West Coast Computer Faire. Exactly how much faster is it than the IBM version?

BRETT: I did some comparative timings when I first got it, my program, at that time, was taking 18 or 19 minutes with the IBM Assembler. With the Microsoft version, it took less than 7 minutes. A factor of 2 and a half. Jeff Garbers, I believe, mentioned a factor of three. This is something everyone should be aware of.

PJ: Which text editor do you use?

BRETT: The Personal Editor.

PJ: Didn't you try some of the removable hard-disk cartridges before you purchased the XT?

BRETT: Yes I did. I had a whole list of complaints before I sent the system back and bought the XT. The removable system was a combination hard disk and removable cartridge setup. The directory was repeatedly clobbered on my hard disk as well as on the cartridge. The manufacturer was a respected major producer of IBM PC compatible hardware, one of the top three. The cartridge was considerably slower than I expected it to be compared to a fixed disk, but the killer was the fact that the removable cartridges could only be read on the original system. If your hard disk went down and you bought a new system, all of your removable backups could be worthless. It also involved a triple boot procedure.

PJ: Do you use a Ramdisk a lot?

BRETT: Yes. My favorite one is Quadram's because it has all the right things. I have a Quadram board on the PC and an AST card in the XT. This keeps me up-to-date with my customers and the market. AST's Ramdisk installs itself in low memory, which I don't like—if you do a warm reboot, you cannot recover what was in memory—whereas Quadram installs their Ramdisk in high memory which allows for easier recovery. Jet Drive is good but doesn't allow you to do DISKCOPY because it does not exactly size the Ramdisk to a double-sided version.

PJ: How do you have your assembler development files set up on your system?

BRETT: I copy everything onto a Ramdisk and work from there. By the way, the Micosoft Assembler comes with a library manager. They used to supply it only with their C compiler.

PJ: Which version of PC-DOS are you running?

BRETT: I use DOS 2.1 on my systems. It has a longer head settle time for floppy disk drives, which is a necessity for the half- height drives. It does slow things down on the XT, but I use floppies so little it makes no substantial difference. I haven't bothered to go and diddle with the disk parameters yet.

Greg Post, Systems Administration, Microsoft

PJ: Greg, what hardware do you use at Microsoft?

GREG: We use a DEC 2060, a DEC 2020, two PDP 1170's, a VAX, and twenty 68000-based systems.

PJ: So most of your software is developed on the larger systems and then transferred to the smaller machines.

GREG: The main reason, of course, is to keep accurate control of the source code. You don't want to have N copies of a program floating around, one for every system you have running.

PJ: Were the development tools that programmers use written inhouse?

GREG: Many of them were. We do use the standard C compiler that came from Bell. Many utilities were supplied that we made liberal modifications to so they would run under our environment.

PJ: What operating systems are you running?

GREG: On the DEC's we are using vanilla TOPS 20. On all the other systems, except for the VAX, we are using XENIX. On the VAX we use Berkeley (UNIX).

PJ: So all of your 68000 systems are using XENIX?

GREG: Right. And everything, except the DEC's, is networked together using Ethernet. We can transfer files back and forth and create virtual terminals on different machines. The other big development tool we use is Electronic Mail. It really adds to the development environment. Sending messages with relevant code is very easy, regardless of whom you wish to reach. You can even participate in group discussions. ❑

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

DATABURST: MAINFRAME-STYLE SCREEN MANAGEMENT

by Steve Cabito, member of the DataBurst development team, The Truchas Grandes Corporation.

OVERVIEW

DataBurst is a tool that allows programmers to design, edit, compile, and display application program screen formats, and then generate source code. The code includes declarations for the screen variables and the calls to the resident runtime support module for keyboard and screen I/O.

In this article, I will describe the system in some detail—what it does, how it works, methods and advantages of using it—and also talk a bit about the development of DataBurst itself.

RATIONALE

DataBurst was designed in an effort to provide the PC with screen management facilities that we were accustomed to using on medium-scale IBM mainframes—specifically, the IBM System/34,/36,/38 family.

Mainframe processors typically reduce programming effort by doing some "front-end" field editing at the terminal. For example, a letter entered into a numeric field will be disallowed by the terminal's firmware; meanwhile, the application program just waits for the I/O to complete. The programmer usually has a good deal of control over field attributes, cursor positioning, enabling of function keys, and scrolling. Screen formats are defined during the development process and are compiled into load modules to be linked by application programs at runtime; constants or literals appearing on screens are compiled into the format modules, not coded into application programs. A collection of screen variables can be thought of as simple data aggregates, no different from record descriptions for a file. DataBurst's screen-processing utilities transfer individual fields to and from their screens and fills the corresponding program variables. Finally, the skeleton source that it generates virtually eliminates any additional screen I/O coding.

DataBurst provides the PC with all of the above facilities, enhances some we thought were difficult to use, and adds a few that we always felt were lacking. Although the original version was designed to be used with the BASIC interpreter and compiler, it has since become language independent by the substitution of different source code generator and language interface modules. I will describe its operation from the viewpoint of the C programmer.

USAGE

Functionally, DataBurst can be thought of as five modules: an on-screen format editor, a format compiler, a verification stub, a source code generator, and the core of the system, an interrupt-driven runtime I/O processor. Briefly, the development process (fig. 1) uses the first four of these modules to create format files and a skeleton source program. The first four modules all use the fifth one. Application logic is then added and the program enters the familiar compile/test/debug phase. The following example details the actual steps followed during the construction of a simple program with two screens. In practice, the more complex the application, the easier it is to develop with DataBurst.

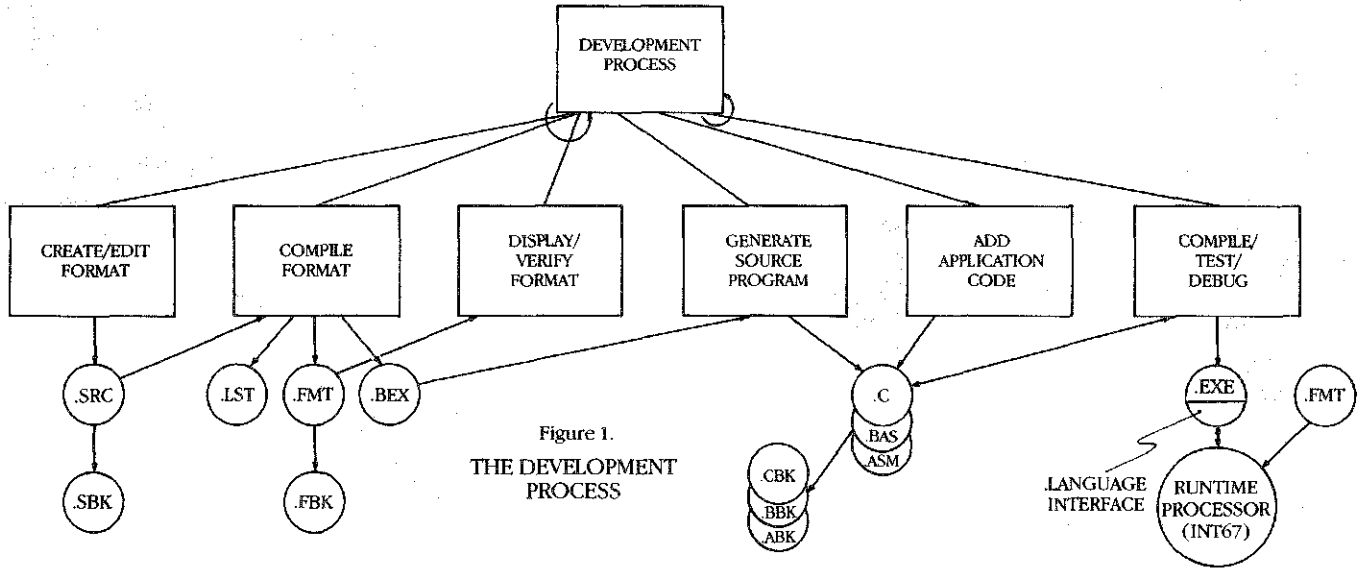


Figure 1.
THE DEVELOPMENT PROCESS

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Let's assume our program performs an inquiry function. It will accept a two-part record key, retrieve the record, and display it. To demonstrate multiple-screen pprrocessing, we will design two formats, the second overlaying the first. The first screen will contain the program title, key input area, message line and function key definitions. The second screen will show the record detail and be situated "within" the first screen.

Option 1 is the screen editor. First, the screens are roughly designed on paper. Then, DataBurst is run and option 1 is selected from the main menu.

DataBurst - Version 1.02
(c) 1983, 1984 Technical Systems Unlimited

1. Create/Update Screen Source File
2. Generate Screen Format File
3. Display Screen Format
4. Generate Source Program
5. Generate Runtime Screen Support

Enter Selection: File Name:

[F9] Execute Selection
[Esc] Exit

Using the editor is a three-step process:

1. Define the screen as it will actually look
2. Supply any [alt]-key substitution strings used during keyboard I/O
3. Specify attributes and variables of both the screens

The first step presents us with a blank screen. We will fill it in as shown below.

```
INVENTORY INQUIRY

Manufacturer: ??????????
Part number :   ????????

------------------------------------------------
Description :  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
On Hand     :  XXXXXX
Back Ordered:  XXXXXX

------------------------------------------------
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

F1-Previous record      F2-Next record      F10-Exit
```

An 'X' denotes an output field, a '?' an I/O field. Anything else is considered constant information. If there is an existing screen definition (.SRC), it will be displayed instead of the blank screen and can be revised (it is first backed up as .SBK). We are then prompted for character strings (0-10 characters) to be assigned, if desired, to [alt]-keys (0-9, A-Z, +, -). Finally, we supply information about the screen,

```
Clear Screen Before Displaying... Y
Flush Keyboard Buffer............ Y

Convert Lowercase to Uppercase... N
Return Ctrl A thru Ctrl Z........ N
Enable F1 thru F10............... Y
Enable Shift F1 thru Shift F10... N
Enable Ctrl F1 thru Ctrl F10..... N
Enable Alt F1 thru Alt F10....... N
```

and additional field descriptions along the lines of:

```
Manufacturer:   ??????????
Part number :     ????????

Field Name......  PARTNO
Line Number..... 04
Start Position... 19
Field Length..... 008
Output Data..... Y
Allow Input..... Y
Constant Type....
Protect Field....
Adjust Fill..... Z
Data Type........ N
Blink Field.....
Reverse Video.... Y
High Intensity...
Non-Display......
Underline........
```

Enter: Continue to the next field
[Esc]: Return to Utility Menu

Most defaults are already displayed. Once this is completed, a condensed parameter file (.SRC) is generated in standard ASCII format. It supplies input for the screen compiler.

Option 2 on the main menu first backs up any existing format file to .FBK, then compiles the screen definition parameters (.SRC) into a runtime format module (.FMT). It also produces a listing from the compiler (.LST) and an extract file (.BEX) used as input to the source code generator. The files are, again, standard ASCII.

Option 3 uses the runtime I/O processor to display the screen as it will appear after a display/read call by the application program. Variables appear as asterisks toaid in verifying placement and length. We can now check that video attributes, numeric justification/fill, [alt]-key strings, etc., are to our liking. If minor changes are required, we can simply edit the .SRC file and continue with option 2.

Option 4 asks you to name each screen you plan to include in the final program. Once this is done, source code is

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

generated from these extract files that declare variable names and lengths. Although DataBurst allows a maximum of 16 simultaneously active screens, it has some memory-management logic which allows a format to be deleted and another to be loaded in its place. (Screen buffers and their contents remain intact between program invocations. This allows advanced applications to pass complete screens and their contents among programs.) Our manufacturer/part number code would look like this:

[Output from the source code generator]

```
/*  name - SAMPLE.C             */
/*  generated 01/01/84 17:07:27 */

/*  DataBurst declarations  */

#include "stdio.h"

#define NUM_FORMATS  2

#define F1           0x3b
#define F2           0x3c
#define F3           0x3d
#define F4           0x3e
#define F5           0x3f
#define F6           0x40
#define F7           0x41
#define F8           0x42
#define F9           0x43
#define F10          0x44
#define ENTER        0x0d

#define DUP_FORMAT   207
#define NOT_LOADED   209

struct screen_1 {
       char *p1_manufact,
            *p1_partno,
            *p1_errmsg; } sample01 = { manufact,
                                       partno,
                                       errmsg };
struct screen_2 {
       char *p2_descript,
            *p2_on_hand,
            *p2_back_ord; } sample02 = { descript,
                                         on_hand,
                                         back_ord };

char *format_name [ ] = { "sample01.fmt  ",
                          "sample02.fmt  " };

/*  screen variables  */

char manufact[10],
     partno[8],
     errmsg[80];
char descript[42],
     on_hand[6],
     back_ord[6];

/*  DataBurst functions  */

db_load()    /* load all formats used */
{
  int p1, p2, i, rtn;

for (i=0; i<NUM_FORMATS; i++) {
  p1 = 4;       /* erase */
  p2 = i + 1;
  if ((rtn = databurst(&p1, &p2, 0)) != 0)
    if (rtn != NOT_LOADED)
      abort("erase failed with %d", p1);

  p1 = 2;       /* load */
  p2 = i + 1;
  if ((rtn = databurst(&p1, &p2, format_name + i)) != 0)
    if (rtn != DUP_FORMAT)
      abort("load failed with %d", p1);
  }
  return;
}

db_display(screen_num)
int screen_num;
{
  int p1 = 3;    /* display a format */

  databurst(&p1, &screen_num, 0);
  if (p1 > 0)
    abort("display failed with %d", p1);
  return;
}

db_write(screen)
char *screen;   /* address of structure of pointers to screen variables *
{
  int p1 = 5;   /* write */

  databurst(&p1, 0, screen);
  if (p1 > 0)
    abort("write failed with %d", p1);
  return;
}

db_read(screen, cursor)
char *screen;
int cursor;     /* field number to position cursor at */
{
  int p1 = 6,
      p2 = cursor;

  databurst(&p1, &p2, screen);
  if (p1 > 0)
    abort("read failed with %d", p1);
  return p2;
}

copy(to, from, len)   /* fixed-length string copy; blank-fills on right *
char *to, *from;
int len;

{
  while (*from && len--)
    *to++ = *from++;
  while (len-- > 0)
    *to++ = ' ';
  return;
}

main()
{


}
```

Notice that the generated program contains a few "extras"—some useful preprocessor declarations and a copy function. Since DataBurst assumes character arrays are to be fixed-length fields, it does not rely upon '+0' for string termination. The copy function acts much as sprintf(to_string, '%.nns', from_string), where nn is the field length. Any unused characters are blank-filled on the right. Functions of the form db_xxxxxxxx() are generated solely for convenience. The runtime processor can always be called directly. Since producing source code involves considerations of style, we tried to strike a compromise between utility and verbosity.

Option 5 is used only with the BASIC interpreter; it generates a BLOAD-format subroutine which issues the interrupts to the runtime processor.

At this point we have a skeleton source program; all we have to do is add application code to the body of main(). I have

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

supposed that our inquiry program will retrieve a record by a two-part key, a manufacturer name (10 characters), and a part number (8 digits). Note that the part number is defined as a numeric zero- filled field. This allows the user to enter 312 as the part number, which becomes 00000312 when the call is completed. Any attempt by the user to enter non-numerics in the part number field will cause the runtime processor to save the 25th line of the screen, display the default message "KBD-01 Field requires numeric characters," lock the keyboard (waiting for a 'reset' key to be pressed), and then restore the 25th line. Similar responses occur for alpha-only fields, disabled function keys, unassigned [alt]-keys, and keys pressed when the cursor has been manually moved outside an input field. The call to read from the screen finally completes when the enter key is pressed.

A note about cursor movement: the tab (shift-tab) key moves the cursor forward (backward) between input fields. The + key at the right of the keypad serves as a "field exit" key. If a non-numeric field is empty, it acts as the tab key does; otherwise, it causes the entry to be justified and blank- or zero-filled as specified. Field editing is possible in much the same manner as with the BASIC editor: Ctrl-cursor right, Ctrl-cursor left, and Ctrl-end are the exact counterparts of the BASIC editor sequences.

The application code (pseudo-code shown within << >>) might then look something like this:

[Application logic added to the skeleton program]

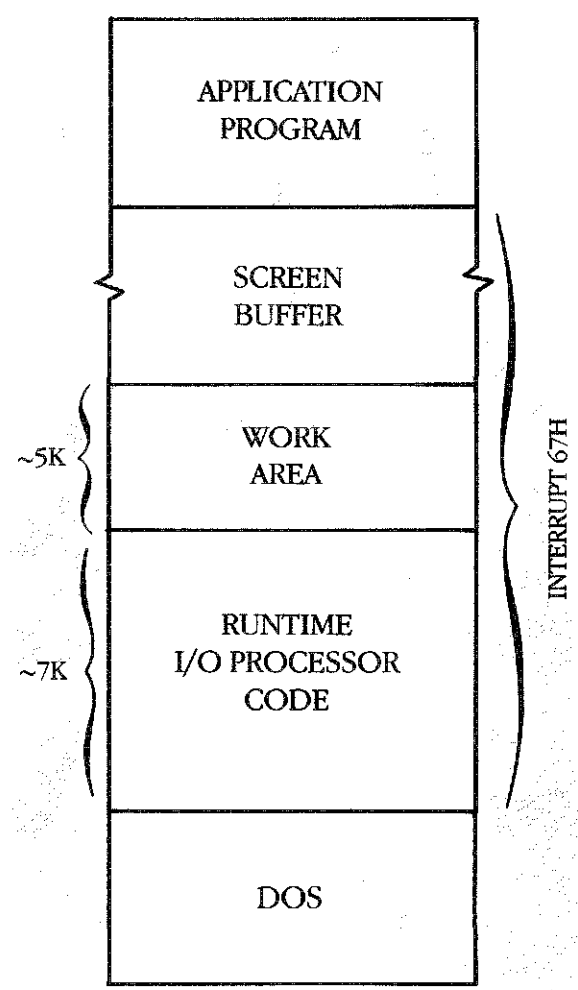
```
int fkey = NULL;

db_load();                      /* load all formats              */

while (fkey != F10) {
  copy(errmsg, " ", 80);        /* clear any messages from screen */
  db_display(1);                /* put up header screen          */
  db_write(&sample01);          /*   and write to it             */
  fkey = db_read(&sample01, 1); /* read from it                  */
  switch (fkey) {               /* key pressed                   */
    case F1:

      << display previous record; break >>

    case F2:

      << display next record; break >>

    case F10:                   /* exit                          */
      break;
    case ENTER:                 /* process the inquiry           */

      << get record key(manufact) length(18) >> /* magic isam function */
      << if(error) copy(errmsg, nasty_message, 80); break; >>

      copy(descript, rec.descr, 42);        /* invisible record buffer */
      copy(on_hand, rec.o_h, 6);
      copy(back_ord, rec.b_o, 6);
      db_display(2);            /* display detail screen         */
      db_write(&sample02);      /* and write detail record       */
      break;
  } /* switch */
} /* while */

/* end of application code */
```

THE RUNTIME PROCESSOR

You will notice that the call to DataBurst is actually to "databurst." This is a public symbol defined in a language- dependent interface module and linked to the application program. It copies the argument addresses (and they MUST be pointers) to a parameter block, adds some other language-specific information to the block, and issues the interrupt to the runtime processor. Pointers are used because occasionally two values are returned. Function calls 6 [read] and 8 [send message] must return the keypress value as well as the status of the operation. We also strove to ensure some degree of generality among languages which do not pass arguments by value.The third parameter in most calls is the address of a structure of character array pointers; this provides DataBurst with addressability of all screen variables. Field lengths are unimportant at this point because they are stored in the .FMT files.

The runtime processor is normally installed at "autoexec time" as int 67H. It occupies 12K of memory, plus a variable amount of screen buffer space (1K-52K, specified during installation). Here is a memory map.

Figure 2: Memory Map

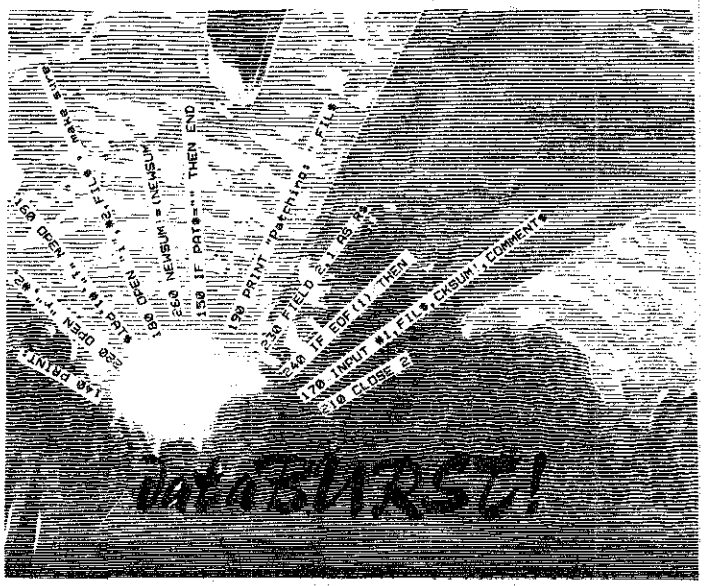


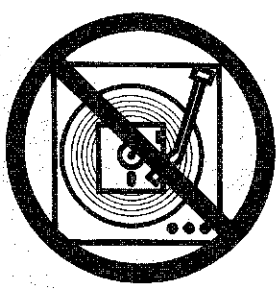
PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

SUMMARY

The development of DataBurst—from the time of inception to the finished product—consumed approximately 2500 man-hours. About two-thirds of the time was directly related to design, programming, and testing. The system currently comprises 144K of code. It was written entirely in assembly language. We presume that in the past year we've learned something about the evolution of a product and how painful it can be at times. Now, just when we'd like to lay back a bit, we are faced with another task, albeit a welcome one: responding to user feedback. It's difficult to describe the momentum gained by a long and encouraging telephone conversation with a total stranger. We were also happy to discover that the entire package runs without modification on the PCjr—it actually improves the performance of the machine, due to our memory-mapped I/O. This implies that the PC can be used as a host development system. All in all, migrating a productive mainframe facility to a microcomputer environment has been a rewarding challenge.

□





Permission Granted

We get letters all the time requesting permission to reprint Fred Crone's Beagle Bros disk envelope warnings in various computer journals. Well, you've got our permission; thanks for asking! Here is a copy of the warnings, suitable for reproduction (unprotected!).

WE ONLY ASK that you print these drawings with both of the credit lines at the bottom. If you don't, and Fred finds out, he'll send his goons out after you.

COPYRIGHT © 1983, BEAGLE BROS INC.
(BEAGLE BROS PUBLISHES APPLE II SOFTWARE.)

BEAGLE BROS / 4315 SIERRA VISTA / SAN DIEGO, CA 92103

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

ASSEMBLY LANGUAGE TUTORIAL

Part II

by Joshua Auerbach

LEARNING THE ASSEMBLER

It is my feeling that many people can teach themselves to use the assembler by reading the MACRO Assembler manual if:

1. You have read and understood a book like Morse and thus have a feeling for the instruction set.

2. You know something about DOS services and so can communicate with the keyboard and screen and do something marginally useful with files. In the absence of this kind of knowledge, you can't write meaningful practice programs and so will not progress.

3. You have access to some good examples (the ones supplied with the assembler are not good, in my opinion. I will try to supply you with some more relevant ones.

4. You ignore the things which are most confusing and least useful. Some of the most confusing aspects of the assembler include the facilities combining segments. You can avoid using all but the simplest of these facilities in many cases, even while writing quite substantial applications.

5. The easiest kind of assembler program to write is a COM program. COM programs might seem harder at first than EXE programs because there is an extra step involved in creating the executable file, but they are structurally very much simpler.

At this point, it is necessary to talk about COM programs and EXE programs. As you probably know, DOS supports two kinds of executable files. EXE programs are much more general, can contain many segments, and are usually built by compilers and sometimes by the assembler. If you follow the lead given by the samples distributed with the assembler, you will end up with EXE programs. A COM program, by contrast, always contains just one segment and receives control with all four segment registers containing the same value. A COM program thus executes, in a simplified environment, a 64K address space. You can go outside this address space simply by temporarily changing one segment register, but you don't have to, and that is what makes COM programs nice and simple. Let's look at a very simple one.

The classic text on writing programs for the C language states that the first thing you should write is a program which says

HELLO, WORLD.

when invoked. What's sauce for C is sauce for assembler, so let's start with a HELLO program of our own. My first presentation of this will be bare bones; not stylistically complete, but just an illustration of what an assembler program absolutely has to have:

```
HELLO   SEGMENT                 ; Set up HELLO code and data section
        ASSUME CS:HELLO,DS:HELLO ; Tell assembler about conditions at entry
        ORG   100H              ; A .COM program begins with 100H byte prefix
MAIN:   JMP   BEGIN             ; Control must start here
MSG     DB    'Hello, world.$'  ; But it is generally useful to put data first
BEGIN:  MOV   DX,OFFSET MSG     ; Let DX --> message
        MOV   AH,9              ; Set DOS function code for printing a message
        INT   21H               ; Invoke DOS
        RET                     ; Return to system
HELLO   ENDS                    ; End of code and data section
        END   MAIN              ; Terminate assembler and specify entry point
```

First, let's attend to some obvious points. The MACRO Assembler uses the general form:

name opcode operands

Unlike the 370 assembler, though, comments are NOT set off from operands by blanks. The syntax uses blanks as delimiters within the operand field (see line 6 of the example), and so all comments must be set off by semi-colons. Line comments are frequently set off with a semi-colon in column 1. I use this approach for block comments, too, although there is a COMMENT statement which can be used to introduce a block comment.

Being an old 370 type, I like to see assembler code in uppercase, although my comments are mixed case. Actually, the assembler is quite happy with mixed case anywhere.

As with any assembler, the core of the opcode set consists of opcodes which generate machine instructions, but there are also opcodes which generate data and ones which function as instructions to the assembler itself, some times called pseudo-ops. In the example, there are five lines which generate machine code (JMP, MOV, MOV, INT, RET), one line which generates data (DB), and five pseudo-ops (SEGMENT, ASSUME, ORG, ENDS, and END). We will discuss all of them.

Now, about labels. You will see that some labels in the example end in a colon and some don't. This is a bit confusing at first, but no real mystery. If a label is attached to a piece of code (as opposed to data), then the assembler needs to know what to do when you JMP to or CALL that label. By convention, if the label ends in a colon, the assembler will use the NEAR form of JMP or CALL. If the label does not end in a colon, it will use the FAR form. In practice, you will always use the colon on any label you are jumping to inside your program because such jumps are always NEAR; there is no reason to use a FAR jump within a single code section. I mention this, though,

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

because omitting the colon isn't usually trapped as a syntax error; it will generally cause something more abstruse to go wrong.

On the other hand, a label attached to a piece of data or a pseudo-op never ends in a colon.

Machine instructions will generally take zero, one, or two operands. Where there are two operands, the one which receives the result goes on the left, as in 370 assembler.

I tried to explain this before; now maybe it will be even clearer. There are many more 8086 machine opcodes than there are assembler opcodes to represent them. For example, there are five kinds of JMP, four kinds of CALL, two kinds of RET, and at least five kinds of MOV depending on how you count them. The MACRO Assembler makes a lot of decisions for you based on the form taken by the operands or on attributes assigned to symbols elsewhere in your program. In the example above, the assembler will generate the NEAR DIRECT form of JMP because the target label BEGIN labels a piece of code instead of a piece of data (this makes the JMP DIRECT) and ends in a colon (this makes the JMP NEAR). The assembler will generate the immediate forms of MOV because the form OFFSET MSG refers to immediate data and because 9 is a constant. The assembler will generate the NEAR form of RET because that is the default and you have not told it otherwise. The DB (define byte) pseudo-op is an easy one: it is used to put one or more bytes of data into storage. There is also a DW (define word) pseudo-op and a DD (define doubleword) pseudo-op; in the PC MACRO Assembler, the fact that a label refers to a byte of storage, a word of storage, or a doubleword of storage can be very significant in ways which we will see presently.

About that OFFSET operator, I guess this is the best way to make the point about how the assembler decides what instruction to assemble: an analogy with 370 assembler:

```
PLACE   DC    ......
        ...
        LA    R1,PLACE
        L     R1,PLACE
```

In 370 assembler, the first instruction puts the address of label PLACE in register 1; the second instruction puts the contents of storage at label PLACE in register 1. Notice that two different opcodes are used. In the PC assembler, the analogous instructions would be:

```
PLACE   DW    ......
        ...
        MOV   DX,OFFSET PLACE
        MOV   DX,PLACE
```

If PLACE is the label of a word of storage, then the second instruction will be understood as a desire to fetch that data into DX. If X is a label, then "OFFSET X" means "the ordinary number which represents X's offset from the start of the segment." And, if the assembler sees an ordinary number, as opposed to a label, it uses the instruction which is equivalent to LA.

If PLACE were the label of a DB pseudo-op, instead of a DW, then

MOV DX,PLACE

would be illegal. The assembler worries about length attributes of its operands.

Next, numbers and constants in general. The assembler's default radix is decimal. You can change this, but I don't recommend it. If you want to represent numbers in other forms of notation such as hex or bit, you generally use a trailing letter. For example,

```
21H       ; is hexidecimal 21,

00010000B ; is the 8 binary number pictured.
```

The next elements we should point to are the SEGMENT... ENDS pair and the END instruction. Every assembler program has to have these elements. SEGMENT tells the assembler you are starting a section of contiguous material (code and/or data). The symmetrically named ENDS statement tells the assembler you are finished with a section of contiguous material. I wish they didn't use the word SEGMENT in this context. To me, a 'segment' is a hardware construct: it is the 64K of real storage which becomes addressable by virtue of having a particular value in a segment register. Now, it is true that the 'segments' you make with the assembler often correspond to real hardware 'segments' at execution time. But if you look at things like the GROUP and CLASS options supported by the linker, you will discover that this correspondence is by no means exact. So, at risk of maybe confusing you even more, I am going to use the more informal term 'section' to refer to the area set off by means of the SEGMENT and ENDS instructions. The sections delimited by SEGMENT... ENDS pairs are really a lot like CSECTs and DSECTs in the 370 world.

```
name    SEGMENT
name    SEGMENT  PUBLIC
name    SEGMENT  AT  nnn
```

I strongly recommend that you be selective in your study of the SEGMENT pseudo-op as described in the manual. Let me just touch on it here.

Basically, you can get away with just the three forms given above. The first form is what you use when you are writing a single section of assembler code which will not be combined

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

with other pieces of code at link time. The second form says that this assembly only contains part of the section; other parts might be assembled separately and combined later by the linker.

I have found that one can construct reasonably large modular applications in assembler simply by making every assembly use the same segment name and declaring the name to be PUBLIC each time. If you read the assembler and linker documentation, you will also be bombarded by information about more complex options such as the GROUP statement and the use of other "combine types" and "classes." I don't recommend getting into any of that. I will talk more about the linker and modular construction of programs a little later. The assembler manual also implies that a STACK segment is required. This is not really true. There are numerous ways to assure that you have a valid stack at execution time.

Of course, if you plan to write applications in assembler which are more than 64K in size, you will need more than what I have told you; but who is really going to do that? Any application that large is likely to be coded in a higher level language.

The third form of the SEGMENT statement makes the delineated section into something like a "DSECT;" that is, it doesn't generate any code, it just describes what is already present somewhere in the computer's memory. Sometimes the AT value you give is meaningful. For example, the BIOS work area is located at location 40 hex. So, you might see

```
BIOSAREA  SEGMENT AT 40H       ; Map BIOS work area
          ORG  BIOSAREA + 10H
EQUIP     DB   ?               ; Location of equipment flags, first byte
BIOSAREA  ENDS
```

in a program which was interested in mucking around in the BIOS work area. At other times, the AT value you give may be arbitrary, as when you are mapping a repeated control block:

```
PROGPREF SEGMENT   AT 0   ; Really a DSECT mapping the program prefix
         ORG   PROGPREF-6
MEMSIZE  DW    ?          ; Size of available memory
PROGPREF ENDS
```

Really, no matter whether the AT value represents truth or fiction, it is your responsibility, not the assembler's, to set up a segment register so that you can really reach the storage in question. So, you can't say

MOV AL,EQUIP

unless you first say something like

```
MOV  AX,BIOSAREA    ; BIOSAREA becomes a symbol with value 40H
MOV  ES,AX
ASSUME ES:BIOSAREA
```

Enough about SEGMENT. The END statement is simple. It goes at the end of every assembly. When you are assembling a subroutine, you just say

END

but when you are assembling the main routine of a program you say

END label

where 'label' is the place where execution is to begin.

Another pseudo-op illustrated in the program is ASSUME. ASSUME is like the USING statement in 370 assembler. However, ASSUME can ONLY refer to segment registers. The assembler uses ASSUME information to decide whether to assemble segment override prefixes and to check that the data you are trying to access is really accessible. In this case, we can reassure the assembler that both the CS and DS registers will address the section called HELLO at execution time. Actually, the SS and ES registers will too, but the assembler never needs to make use of this information.

I guess I have explained everything in the program except that ORG pseudo-op. ORG means the same thing as it does in many assembly languages. It tells the assembler to move its location counter to some particular address. In this case, we have asked the assembler to start assembling code hex 100 bytes from the start of the section called HELLO instead of at the very beginning. This simply reflects the way COM programs are loaded. When a COM program is loaded by the system, the system sets up all four segment registers to address the same 64K of storage. The first 100 hex bytes of that storage contains what is called the program prefix; this area is described in appendix E of the DOS manual. Your COM program physically begins after this. Execution begins with the first physical byte of your program; that is why the JMP instruction is there.

Wait a minute, you say, why the JMP instruction at all? Why not put the data at the end? Well, in a simple program like this I probably could have gotten away with that. However, I have the habit of putting data first and would encourage you to do the same because of the way the assembler has of assembling different instructions based on the nature of the operand.

Unfortunately, sometimes the different choices of instruction which can assemble from a single opcode have different lengths. If the assembler has already seen the data when it gets to the instructions it has a good chance of reserving the right number of bytes on the first pass. If the data is at the end, the assembler may not have enough information on the first pass to reserve the right number of bytes for the instruction. Sometimes the assembler will complain about this, something like 'forward reference is illegal,' but at other times it will make some default assumption. On the second pass, if the assumption turned out to be wrong, it will report what is called a "Phase error," a very nasty error to track down. So get

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

in the habit of putting data and equated symbols ahead of code.

OK. Maybe you understand the program now. Let's walk through the steps involved in making it into a real COM file.

1. The file should be created with the name HELLO.ASM (actually the name is arbitrary but the extension .ASM is conventional and useful)

2. ASM HELLO,,;

(This is just one example of invoking the assembler. It uses the small assembler ASM, and it produces an object file and a listing file with the same name as the source file. I am not going exhaustively into how to invoke the assembler, which the manual goes into pretty well. I guess this is the first time I mentioned that there are really two assemblers; the small assembler ASM will run in a 64K machine and doesn't support macros. I used to use it all the time; now that I have a bigger machine and a lot of macro libraries, I use the full-function assembler MASM. You get both when you buy the package.)

3. If you issue DIR at this point, you will discover that you have acquired HELLO.OBJ (the object code resulting from the assembly) and HELLO.LST (a listing file). I guess I can digress for a second here concerning the listing file. It contains TAB characters. I have found there are two good ways to get it printed and one bad way. The bad way is to use LPT1: as the direct target of the listing file or to try copying the LST file to LPT1 without first setting the tabs on the printer. The two good ways are:

a. Direct it to the console and activate the printer with CTRL-PRTSC. In this case, DOS will expand the tabs for you.

b. Direct it to LPT1: but first send the right escape sequence to LPT1 to set the tabs every eight columns. I have found that on some early serial numbers of the IBM PC printer, tabs don't work quite right, which forces you to the first option.

4. LINK HELLO;

(Again, there are lots of linker options but this is the simplest. It takes HELLO.OBJ and makes HELLO.EXE.) HELLO.EXE? I thought we were making a COM program, not an EXE program. Right. HELLO.EXE isn't really executable; it's just that the linker doesn't know about COM programs. That requires another utility. You don't have this utility if you are using DOS 1.0; you have it if you are using DOS 1.1 or DOS 2.0. Oh, by the way, the linker will warn you that you have no stack segment. Don't worry about it.

5. EXE2BIN HELLO HELLO.COM

This is the final step. It produces the actual program you will execute. Note that you have to spell out HELLO.COM; for a nominally rational but actually perverse reason; EXE2BIN uses the default extension BIN instead of COM for its output file. At this point, you might want to erase HELLO.EXE; it looks a lot more useful than it is. Chances are you won't need to recreate HELLO.COM unless you change the source, and then you are going to have to redo the whole thing.

6. HELLO

You type hello, that invokes the program, it says

HELLO YOURSELF!!

(oops, what did I do wrong...?)

WHAT ABOUT SUBROUTINES?

I started with a simple COM program because I actually think they are easier to create than subroutines to be called from high=level languages, but maybe it's really the latter you are interested in. Here, I think you should get comfortable with the assembler FIRST, with little exercises like the one above and also another one which I will finish up with. Next, you are ready to look at the interface information for your particular language. You usually find this in some sort of appendix. For example, the BASIC manual has appendix C on Machine Language Subroutines. The PASCAL manual buries the information a little more deeply: the interface to a separately compiled routine can be found in the chapter on Procedures and Functions, in a subsection called Internal Calling Conventions. Each language is slightly different, but here are what I think are some common issues in subroutine construction.

1. NEAR versus FAR? Most of the time, your language will probably call your assembler routine as a FAR routine. In this case, you need to make sure the assembler will generate the right kind of return. You do this with a PROC...ENDP statement pair. The PROC statement is probably a good idea for a NEAR routine too, even though it is not strictly required:

```
        FAR linkage:                    NEAR linkage:

ARBITRARY SEGMENT                   SPECIFIC  SEGMENT  PUBLIC
          PUBLIC THENAME                      PUBLIC THENAME
          ASSUME CS:ARBITRARY                 ASSUME CS:SPECIFIC,DS:SPECIFIC
THENAME   PROC FAR                            ASSUME ES:SPECIFIC,SS:SPECIFIC
          ..... code and data       THENAME   PROC NEAR
THENAME   ENDP                                ..... code and data ....
ARBITRARY ENDS                      THENAME   ENDP
          END                       SPECIFIC  ENDS
                                              END
```

With FAR linkage, it doesn't really matter what you call the segment. You must declare the name by which it will be called in a PUBLIC pseudo-op and also show that it is a FAR procedure. Only CS will be initialized to your segment when you are called. Generally, the other segment registers will continue to point to the caller's segments. With NEAR linkage, you are executing in the same segment as the caller. Therefore, you must give the segment a specific name as instructed

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

by the language manual. However, you may be able to count on all segment registers pointing to your own segment (sometimes the situation can be more complicated but I cannot really go into all of the details). You should be aware that the code you write will not be the only thing in the segment and that it will be physically relocated within the segment by the linker. However, all OFFSET references will be relocated and will be correct at execution time.

```
ARGS    STRUC
        DW   3 DUP(?)  ; Saved BP and return address
ARG3    DW   ?
ARG2    DW   ?
ARG1    DW   ?
ARGS    ENDS
     ..........
        PUSH BP                 ; save BP register
        MOV  BP,SP              ; Use BP to address stack
        MOV   ...,[BP].ARG2     ; retrieve second argument
        (etc.)
```

2. Parameters passed on the stack. Usually, high-level languages pass parameters to subroutines by pushing words onto the stack prior to calling you. What may differ from language to language is the nature of what is pushed (OFFSET only or OFFSET and SEGMENT) and the order in which it is pushed (left to right, right to left within the CALL statement). However, you will need to study the examples to figure out how to retrieve the parameters from the stack. A useful fact to exploit is that a reference involving the BP register defaults to a reference to the stack segment. So, the following strategy can work:

This example uses something called a structure, which is only available in the large assembler; furthermore, it uses it without allocating it, which is not a well-documented option. However, I find the above approach generally pleasing. The STRUC is like a DSECT in that it establishes labels as being offset a certain distance from an arbitrary point. These labels are then used in the body of code by beginning them with a period; the construction ".ARG2" means, basically, " + (ARG2-ARGS)."

What you are doing here is using BP to address the stack, accounting for the word where you saved the caller's BP and also for the two words which were pushed by the CALL instruction.

3. How big is the stack? BASIC only gives you an eight word stack to play with. On the other hand, it doesn't insist that you save any registers except the segment registers. Other languages give you a liberal stack, which makes things a lot easier. If you have to create a new stack segment for yourself, the easiest way is to place the stack at the end of your program and:

```
CLI                     ; suppress interrupts while changing stack
MOV  SSAVE,SS           ; save old SS in local storage (old SP
                        ;  already saved in BP)
MOV  SP,CS              ; switch
MOV  SS,SP              ; the
MOV  SP,OFFSET STACKTOP ; stack
STI                     ; (maybe)
```

Later, you can reverse these steps before returning to the caller. At the end of your program, you place the stack itself:

```
DW   128 DUP(?)          ; stack of 128 words (liberal)
STACKTOP LABEL WORD
```

4. Make sure you save and restore those registers required by the caller.

5. Be sure to get the right kind of addressability. In the FAR call example, only CS addresses your segment. If you are careful with your ASSUME statements the assembler will keep track of this fact and generate CS prefixes when you make data references; however, you might want to do something like

```
MOV AX,CS       ; get current segment address
MOV DS,AX       ; To DS
ASSUME DS:THISSEG
```

Be sure you keep your ASSUMEs in synch with reality.

LEARNING ABOUT BIOS AND THE HARDWARE

You can't do everything with DOS calls. You may need to learn something about the BIOS and about the hardware itself. In this, the Technical Reference is a very good thing to look at.

The first thing you look at in the Technical Reference, unless you are really determined to master the whole ball of wax, is the BIOS listings presented in Appendix A. Glory be: here is the whole 8K of ROM which deals with low=level hardware support laid out with comments and everything. In fact, if you are just interested in learning what BIOS can do for you, you just need to read the header comments at the beginning of each section of the listing.

BIOS services are invoked by means of the INT instruction; the BIOS occupies interrupts 10H through 1FH and also interrupt 5H; actually, of these 17 interrupts, five are used for user exit points or data pointers, leaving 12 actual services.

In most cases, a service deals with a particular hardware interface; for example, BIOS interrupt 10H deals with the screen. As with DOS function calls, many BIOS services can be passed a function code in the AH register and possible other arguments.

I am not going to summarize the most useful BIOS features here; you will see some examples in the next sample program we will look at. The other thing you might want to get into with the Tech Reference is the description of some hardware options, particularly the asynch adapter, which are not well supported in the BIOS. The write up on the asynch adapter is pretty complete.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Actually, the Tech Reference itself is pretty complete and very nice as far as it goes. One thing which is missing from it is information on the programmable peripheral chips on the system board. These include

the 8259 interrupt controller
the 8253 timer
the 8237 DMA controller
the 8255 peripheral interface

To make your library absolutely complete, you should order the INTEL data sheets for these beasts.

I should say, though, that the only one I ever found I needed to know about was the interrupt controller. If you happen to have the 8086 Family User's Manual, the big book put out by INTEL, which is one of the things people sometimes buy to learn about 8086 architecture, there is an appendix which gives an adequate description of the 8259.

A FINAL EXAMPLE

I leave you with a more substantial example of code which illustrates some good elementary techniques; I won't claim its style is perfect, but I think it is adequate. I think this is a much more useful example than what you will get with the assembler:

```
        PAGE 61,132
        TITLE SETSCRN -- Establish correct monitor use at boot time

;    This program is a variation on many which toggle the equipment flags
; to support the use of either video option (monochrome or color).
; The thing about this one is it prompts the user in such a way that he
; can select the use of the monitor he is currently looking at (or which
; is currently connected or turned on) without really having to know
; which is which. SETSCRN is a good program to put first in an
; AUTOEXEC.BAT file.
;    This program is highly dependent on the hardware and BIOS of the IBM PC
; and is hardly portable, except to very exact clones. For this reason,
; BIOS calls are used in lieu of DOS function calls where both provide
; equal function.
;
```

OK. That's the first page of the program. Notice the PAGE statement, which you can use to tell the assembler how to format the listing. You give it lines per page and characters per line. I have mine set up to print on the host line printer; I routinely upload my listings at 9600 baud and print them on the host; it is faster than using the PC printer. There is also a TITLE statement. This simply provides a nice title for each page of your listing. Now for the second page:

```
        SUBTTL -- Provide .COM type environment and Data
        PAGE
;
;       First, describe the one BIOS byte we are interested in
;
BIOSDATA  SEGMENT   AT 40H      ; Describe where BIOS keeps his data
          ORG       10H         ; Skip parts we are not interested in
EQUIP     DB        ?           ; Equipment flag location
MONO      EQU       00110000B ; These bits on if monochrome
COLOR     EQU       11101111B ; Mask to make BIOS think of the color board
BIOSDATA  ENDS                  ; End of interesting part
;
;       Next, describe some values for interrupts and functions
;
DOS       EQU       21H         ; DOS Function Handler INT code
PRTMSG    EQU       09H         ; Function code to print a message
KBD       EQU       16H         ; BIOS keyboard services INT code
GETKEY    EQU       00H         ; Function code to read a character
SCREEN    EQU       10H         ; BIOS Screen services INT code
MONOINIT  EQU       02H         ; Value to initialize monochrome screen

;COLORINIT EQU       03H        ; Value to initialize color screen (80x25)
COLORINIT EQU      01H         ; Value to initialize color screen (40X25)
;
;       Now, describe our own segment
;
SETSCRN   SEGMENT             ; Set operating segment for CODE and DATA
;
          ASSUME CS:SETSCRN,DS:SETSCRN,ES:SETSCRN,SS:SETSCRN    ;All segments
;
          ORG       100H      ; Begin assembly at standard .COM offset
;
MAIN      PROC      NEAR      ; COM files use NEAR linkage
          JMP       BEGIN     ; And, it is helpful to put the data first, but
;                             ; then you must branch around it.
;
;         Data used in SETSCRN
;
CHANGELOC   DD      EQUIP     ;Location of the EQUIP, recorded as far pointer
MONOPROMPT  DB      'Please press the plus (+) key.$'    ; User sees on mono
COLORPROMPT DB      'Please press the minus (-) key.$'   ; User sees on color
```

Several things are illustrated on this page. First, in addition to titles, the assembler supports subtitles: hence the SUBTTL pseudo-op. Second, the PAGE pseudo-op can be used to go to a new page in the listing. You see an example here of the DSECT-style segment in the "SEGMENT AT 40H". Here, our our interest is in correctly describing the location of some data in the BIOS work area which really is located at segment 40H.

You will also see illustrated the EQU instruction, which just gives a symbolic name to a number. I don't make a fetish of giving a name to every single number in a program. I do feel strongly, though, that interrupts and function codes, where the number is arbitrary and the function being performed is the thing of interest, should always be given symbolic names.

One last new element in this section is the define doubleword (DD) instruction. A doubleword constant can refer, as in this case, to a location in another segment. The assembler will be happy to use information at its disposal to properly assemble it. In this case, the assembler knows that EQUIP is offset 10 in the segment BIOSDATA which is at 40H.

```
        SUBTTL -- Perform function
        PAGE
BEGIN:  CALL    MONOON                  ; Turn on mono display
        MOV     DX,OFFSET MONOPROMPT     ; GET MONO PROMPT
        MOV     AH,PRTMSG                ; ISSUE
        INT     DOS                      ; IT
        CALL    COLORON                 ; Turn on color display
        MOV     DX,OFFSET COLORPROMPT    ; GET COLOR PROMPT
        MOV     AH,PRTMSG                ; ISSUE
        INT     DOS                      ; IT
        MOV     AH,GETKEY               ; Obtain user response
        INT     KBD
        CMP     AL,'+'                  ; Does he want MONO?
        JNZ     NOMONO

        CALL    MONOON                  ; yes, give it to him
NOMONO: RET
MAIN    ENDP
```

The main code section makes use of subroutines to keep the basic flow simple. About all that's new to you in this section is the use of the BIOS interrupt KBD to read a character from the keyboard.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Now for the subroutines, MONOON and COLORON:

```
        SUBTTL -- Routines to turn monitors on
        PAGE
MONOON  PROC    NEAR                    ; Turn mono on
        LES     DI,CHANGELOC            ; Get location to change
        ASSUME  ES:BIOSDATA             ; TELL ASSEMBLER ABOUT CHANGE TO ES
        OR      EQUIP,MONO
        MOV     AX,MONOINIT             ; Get screen initialization value
        INT     SCREEN                  ; Initialize screen
        RET
MONOON  ENDP
COLORON PROC    NEAR                    ; Turn color on
        LES     DI,CHANGELOC            ; Get location to change
        ASSUME  ES:BIOSDATA             ; TELL ASSEMBLER ABOUT CHANGE TO ES
        AND     EQUIP,COLOR
        MOV     AX,COLORINIT            ; Get screen initialization value
        INT     SCREEN                  ; Initialize screen
        RET
COLORON ENDP
SETSCRN ENDS                            ; End of segment
        END     MAIN                    ; End of assembly; execution at MAIN
```

The instructions LES and LDS are useful ones for dealing with doubleword addresses. The offset is loaded into the operand register and the segment into ES (for LES) or DS (for LDS). By telling the assembler, with an ASSUME, that ES now addresses the BIOSDATA segment, it is able to correctly assemble the OR and AND instructions which refer to the EQUIP byte. An ES segment prefix is added.

To understand the action here, you simply need to know that flags in that particular byte control how the BIOS screen service initializes the adapters. BIOS will only work with one

Joshua Auerbach developed this tutorial while working as a programmer for the Yale University Computing Center. He is now working for the IBM Research Division, Thomas J. Watson Research Staff Member, Yorktown Heights, NY.

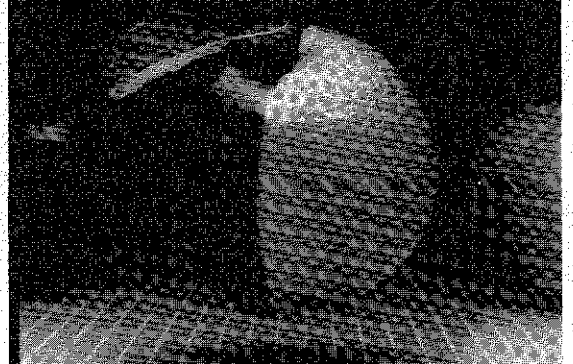
This paper was part of the Integrated Personal Computers Project, Installation Code YU, Communications Group, Communications and Data Base Division, Session C316.

The article was reprinted courtesy of SHARE, Inc.Copyright (C) 1984 by SHARE, Inc. Published by SHARE, Inc, a nonprofit Delaware membership corporation, permission to reprint in whole or in part may be granted for educational or scientific purposes upon written application to the secretary of:

SHARE, Inc
1 Illinois Center
111 E. Wacker Dr.
Chicago, IL 60601

□

NOW YOU CAN WORK IN C WITH ONLY BASIC KNOWLEDGE



Enjoy the Best of Both Worlds with BASIC_C™!

Write or Call Collect

SOURCE
C Source
12801 Frost Rd.
Kansas City, MO 64138
816-353-8808

Until now, IBM BASIC and C were about as far apart as apples and oranges — and if you wanted to move from BASIC to C, you had to go back to square one. Not anymore!

There's no denying that BASIC is a fine language: it has probably helped you learn a lot about programming computers. But now that you know how to program, isn't it time to take the next step — to a language that was designed to help you quickly develop software that is fast, compact, and portable?

BASIC_C can help you make the transition smoothly: it's a library of C functions providing BASIC's high level capabilities you've come to know so well. Functions for string handling, sequential and random i/o, graphics, and many more — all designed to look and work like BASIC.

You get the source code to BASIC_C: extensively documented, it provides many examples to help you learn how to write C functions, and if a function doesn't happen to suit your specific needs . . . change it!

C Source provides a complete development package: the BASIC_C Library, a full screen editor, and a choice of three C compilers. Buy it all, or just the pieces you need.

BASIC_C Library - source code	$175
EC - full screen editor	$ 85
BASIC_C & EC	$225
BASIC_C & DeSmet C Compiler	$275
BASIC_C & Computer Innovations C compiler	$475
BASIC_C & Lattice C compiler	$550
BASIC_C, DeSmet C Compiler, & EC	$325
BASIC_C, Computer Innovations C compiler, & EC	$525
BASIC_C, Lattice C compiler, & EC	$600

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

GRAPHIC WINDOWS

by Ronald Moak

Including graphics in a program can greatly enhance the application. Graphics can reinforce the link between a computer user and a column of data or an equation. This visual emphasis can mean the difference between comprehension and confusion. But what may be enjoyable for the user to see can be excruciatingly painful for the programmer to create.

A program should have complete control over gathering and displaying data. When it comes to graphing information, the programmer often has little idea of the numeric boundaries he will face. A graphic display quickly loses its appeal when it begins to wander all over the screen, destroying carefully laid- out text. Complete control must be achieved regardless of the data being displayed.

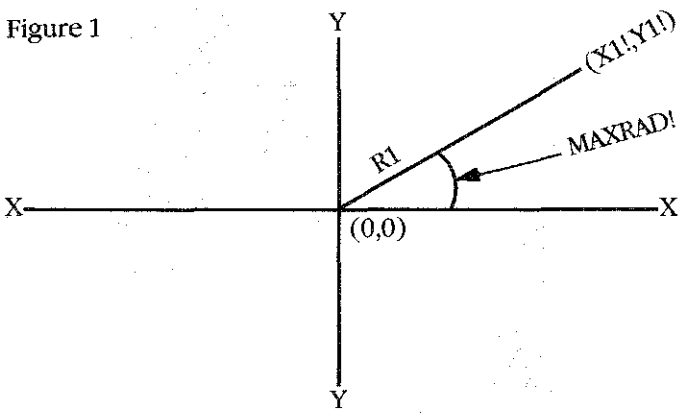
The program we will discuss illustrates one technique for plotting a graph without rigidly fixed limits. The user defines a window for a graph, and the program draws the lines within the window. The location and size of the window can then be changed, and the graph can always be redrawn to fit the window.

There is a simple technique for controlling the plotting of unknown data. First, run the data through an equation to find the max/min limits of the information. These figures will be combined with the physical limits of the window to form a ratio. The ratio is then used to locate the dots on the screen.

THE PROGRAM

In our listing, the equations in subroutine 2000 are called in the first WHILE loop. The first equation calculates a radius, given a constant and a radian. The second two equations calculate the values of X! and Y!. The radius is the distance between (X!,Y!) and (0,0) on a coordinate graph. Because the maximum and minimum of X! and Y! are used to calculate the limits of the graph, we trap them in subroutine 2100.

Figure 1



Once we have calculated the absolute maximum and minimum for X! and Y!, we can call subroutine 2300 to calculate the ratios of the physical limits divided by the calculated limits:

X.RATIO! = (X2! - X1!) / (X.MAX! - X.MIN!)

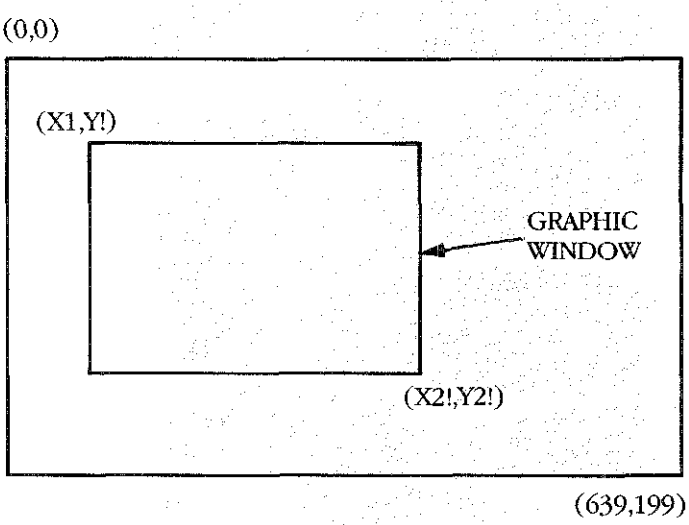
X2! and X1! represent the maximum and minimum physical limits of the defined window on the X axis. The figure X.RATIO! represents the ratio of the physical and calculated limits. For example, on a high-resolution screen with a user defined window of (20,20) - (500, 180), X1! equals 20, and X2! equals 500. If the calculated X.MAX! and X.MIN! for the equation was 3 and -3 respectively, the resulting ratio would be 80.

X.RATIO! = (X2! - X1!) / (X.MAX! - X.MIN!)

80 = (500 - 20) / (3 - (-3))

In other words, every change in X! of 1 would result in a change of 80 pixels on the screen. The Y.RATIO! works in exactly the same way as the X.RATIO!

Figure 2



The next process is to recalculate the equation. If the length of time it takes to recalculate things is a problem, the results of the first calculation can be placed in an array to speed it up. Using the ratio, we can calculate the physical location on the screen where the points are plotted. The calculated values of X! and Y! are passed to subroutine 2400. In subroutine 2400 the calculation of (X! - X.MIN!) determines the relative position of X! on the line defined X.MAX! - X.MIN!:

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

X.MIN!_____X!_____X.MAX!

The relative position of X! is multiplied by the ratio X.RATIO! to get the physical offset from the left margin of the screen. This offset is then added to X1! to place the point within the borders of the window.

Continuing with the previous example, if the calculated value of X! were 1, the value would be 4, coming from (1 - (-3)). Multiply 4 by the ratio X.RATIO!, or 80, and you get the offset value of 320. Add 320 to X1!, or 20 and you get X.PLOT! = 340.

X.PLOT! = X1! + (X.RATIO! * (X! - X.MIN!))

340 = 20 + (80 * (1 - (-3))

The Y.PLOT! calculation is slightly different from the X.PLOT!, due to the way IBM addresses screen graphics. In a normal (X,Y) coordinate system, the position (0,0) is in the lower left-hand corner when the values for X and Y are positive. IBM, on the other hand, placed the (0,0) in the upper left-hand corner (figure 2). If we added the physical offset to the Y1! value, the graph we would generate would be inverted, compared to a graph plotted on the standard coordinate system. To correct the problem, we subtract the offset from the maximum Y border of the window. This will result in a normal-looking graph.

The variables in the equation may be changed interactively to create a number of interesting graphic shapes. Experimenting on your own will produce some very exciting pictures. For the first three variables (constant, maximum, and # of degrees,) try 3.7865, 36000, and 188.5 or maybe 2.7865, 54000, and 197.5565.

THE MOST EXTENSIVE C LIBRARY ANYWHERE

THE GREENLEAF FUNCTIONS
Library for C Programmers
Total Access to IBM PC and PC XT
Fully Compatible with CI C86
Lattice and Microsoft C,
DOS 2.0 and DOS 1.1

A GOOD WAY TO LEARN C

FULL SOURCE CODE

PARTIAL CONTENTS

◆ DOS 2.0 – over 25 functions ◆ Complete Video Access for Text and Graphics ◆ Over 60 String Functions (including "justify" and "center") ◆ Rainbow Series Color Text ◆ Time and Date ◆ Over 40 IBM Graphics Printer functions ◆ Total Access to Function and Special Keys ◆ RS232 Async ◆ All BIOS Functions ◆ Some functions are "BASIC Equivalent" ◆ Software Diagnostics ◆ Disk functions ◆ Utility functions ◆ and more . . .

THE GREENLEAF FUNCTIONS . . .
Nearly 200 functions, 180 page manual, 3 Libraries, Extensive Examples of each function, Full Source Code in C and Assembler, Demo Programs

Save Months of Painstaking Work . . .

$175.00
Add $7.00 for postage and handling. Specify Compiler – (C86, Lattice, or Microsoft)
MasterCard and VISA Accepted
Price subject to change without notice.
Dealer Inquiries Welcome
(214) 446-8641

GREENLEAF SOFTWARE ©

FOR THE IBM PC AND PC XT

The Greenleaf Functions is a trademark of Greenleaf Software. C86 is a trademark of Computer Innovations Inc. IBM, IBM PC, PC XT, and PCDOS are trademarks of International Business Machines Corp. Lattice is a trademark of Lattice Inc. Microsoft C and MSDOS are trademarks of Microsoft Inc.

GREENLEAF SOFTWARE ◆ 2101 HICKORY DRIVE ◆ CARROLLTON, TEXAS 75006

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
'┌──────────────────────────────────────────────────────────┐
'│                                                          │
'│                     Graphics Program                     │
'│                                                          │
'│      Program plots X & Y on the screen using a standard  │
'│    formula.  The points are plotted within a user defined│
'│    window.                                               │
'│                                                          │
'│                  (C) Copyright 1984                      │
'│                                                          │
'│                     Ronald Moak                          │
'└──────────────────────────────────────────────────────────┘

   MAXANG! = 3680 : DINC! = 10: CON! = 1.9    ' Program defaults upon
   X1! = 0: X2! = 639: Y1! = 0: Y2! = 199     ' start up

1000 CLS: SCREEN 2,0,0
     LOCATE 1,1: PRINT
"   This Graphics routine generates a picture based upon the formula:     "
"                                                                         "
"                        R = COSINE(Constant) * Degrees                   "
"                                                                         "
"      The picture is displayed in the Window defined by the locations    "
"  (X1!,Y1!) upper left corner and (X2!,Y2!) lower right corner of window."

     LOCATE  8,7 : PRINT "Enter Constant for the formula    default: " CON!
     LOCATE  9,7 : PRINT "Enter maximum number of degrees   default: "
MAXANG!
     LOCATE 10,7 : PRINT "Enter # degrees of increment      default: " DINC!
     LOCATE 12,7 : PRINT "              Define Screen Window "
     LOCATE 14,7 : PRINT "Enter Upper Left X value  (X1!)   default: " X1!
     LOCATE 15,7 : PRINT "Enter Upper Left Y value  (Y1!)   default: " Y1!
     LOCATE 16,7 : PRINT "Enter Lower Right X value (X2!)   default: " X2!
     LOCATE 17,7 : PRINT "Enter Lower Right Y value (Y2!)   default: " Y2!
     LOCATE 20,5 : PRINT "Enter the new value or return for the default
value."

     LOCATE  8,60: INPUT; SCON!          '
     LOCATE  9,60: INPUT; SMAXANG!       '   Input values
     LOCATE 10,60: INPUT; SDINC!         '
     LOCATE 14,60: INPUT; SX1!           '
     LOCATE 15,60: INPUT; SY1!           '
     LOCATE 16,60: INPUT; SX2!           '
     LOCATE 17,60: INPUT; SY2!           '

     IF SMAXANG! <> 0 THEN MAXANG! = SMAXANG!  '  Test to check if the
     IF SCON!  <> 0 THEN CON!  = SCON!         '  input value was changed.
     IF SDINC! <> 0 THEN DINC! = SDINC!        '  Replace the default with
     IF SX1!  <> 0 THEN  X1!  = SX1!           '  new value.
     IF SY1!  <> 0 THEN  Y1!  = SY1!           '
     IF SX2!  <> 0 THEN  X2!  = SX2!           '
     IF SY2!  <> 0 THEN  Y2!  = SY2!           '

     LOCATE 23,20: PRINT "      Calculating First Pass";

     X.MAX! = -100000 : X.MIN! = 100000  '  Initalize Variables
     Y.MAX! = -100000 : Y.MIN! = 100000  '
     MAXRAD! = .017453 * MAXANG!         ' Convert Maximun degrees to radians
     INC! = .017453 * DINC!              ' Convert Increment degrees to
radians
     RAD! = 0                            '
     WHILE RAD! < MAXRAD!          '
        GOSUB 2000                 ' Calculate the value of X! & Y!
        GOSUB 2100                 ' Save the max & min value X! & Y!
        RAD! = RAD! + INC          '
     WEND

     CLS                           ' Clear Screen
     GOSUB 2300                    ' Calculate ratio
     RAD! = 0:  V% = 0             '
     LINE (X1!,Y1!)-(X2!,Y2!),,B   ' Print border

     WHILE RAD! < MAXRAD!          '
        GOSUB 2000                 ' Calculate the value of X! & Y!
        GOSUB 2400                 ' Calculate X.PLOT! & Y.PLOT! values
        GOSUB 2200                 ' Plot values on screen
        RAD! = RAD! + INC          '
     WEND

     LOCATE 25,1: PRINT "Continue <y/n>";       ' Pause with option to
1100 I$ = INKEY$: IF I$ = "" THEN 1100          ' end program
     IF I$ = "N" OR I$ = "n" THEN END ELSE 1000 '
'┌──────────────────────────────────────────────────────────┐
'│                                                          │
'│      Equation used to calculate X & Y                    │
'│                                                          │
'└──────────────────────────────────────────────────────────┘

2000 R! = COS((CON!) * RAD!)    ' Calculate radius
     X! = R! * COS(RAD!)        ' Calculate X!
     Y! = R! * SIN(RAD!)        ' Calculate Y!
     RETURN
'┌──────────────────────────────────────────────────────────┐
'│                                                          │
'│     Trap for max & min value for X & Y                   │
'│                                                          │
'└──────────────────────────────────────────────────────────┘

2100 IF X! > X.MAX! THEN X.MAX! = X!
     ELSE IF X! < X.MIN! THEN X.MIN! = X!

     IF Y! > Y.MAX! THEN Y.MAX! = Y!
     ELSE IF Y! < Y.MIN! THEN Y.MIN! = Y!
     RETURN
'┌──────────────────────────────────────────────────────────┐
'│                                                          │
'│     Plot X & Y on the screen                             │
'│                                                          │
'└──────────────────────────────────────────────────────────┘

2200 IF V% THEN LINE -(X.PLOT!,Y.PLOT!)
     ELSE PRESET (X.PLOT!,Y.PLOT!): V% = 1
     RETURN
'┌──────────────────────────────────────────────────────────┐
'│                                                          │
'│     Calculate the ratio between screen limits and        │
'│         X & Y calculated limits                          │
'│                                                          │
'└──────────────────────────────────────────────────────────┘

2300 X.RATIO! = (X2! - X1!) / (X.MAX! - X.MIN!)
     Y.RATIO! = (Y2! - Y1!) / (Y.MAX! - Y.MIN!)
     RETURN
'┌──────────────────────────────────────────────────────────┐
'│                                                          │
'│     Calculate the X & Y Point to plot                    │
'│                                                          │
'└──────────────────────────────────────────────────────────┘

2400 X.PLOT! = X1! + (X.RATIO! * (X! - X.MIN!))
     Y.PLOT! = Y2! - (Y.RATIO! * (Y! - Y.MIN!))
     RETURN
```

□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

THE PROGRAMMER'S SHOP

helps compare, evaluate, find products. Straight answers for serious programmers.

SERVICES

- Programmer's Referral List
- Compare Products
- Help find a Publisher
- Evaluation Literature free
- BULLETIN BOARD - 7 PM to 7 AM 617-461-0174
- Dealer's Inquire
- Newsletter
- Rush Order
- Over 300 products

Our Free Report: PRODUCTIVITY - MSDOS

Assume use of compiler and typical editor. What commercial or public domain products, what techniques improve productivity? "Productivity with MSDOS" is a growing document with some answers. Call to request it. **Help improve it. Earn $50** credit toward any purchase when we add any description, code, or idea received from you.

RECENT DISCOVERIES

PROFILER - Examine MSDOS program execution speeds. Determine where to improve programs in any Microsoft language, Lattice, or C86. Make histograms that show time spent in portions of your program, and doing MSDOS I/O, etc. $175

"C" LANGUAGE

		LIST PRICE	OUR PRICE
APPLE: AZTEC C - Full, ASM		$199	call
8080: BDS C - Fast, popular		150	125
8080: AZTEC C - Full		199	call
Z80: ECOSOFT - Fast, Full		250	225
8086: C86 - optimizer, Meg		395	call
8086: Lattice - New 1.1 & 2.0		500	call
Microsoft (Lattice)	MSDOS	500	call
Digital Research - Megabyte	8086	350	269
Desmet by CWare - Fast	8086	109	99

BASIC

	ENVIRONMENT		
Active Trace - debug	8080/86	$ 80	72
MBASIC-80 - MicroSoft	8080	375	255
BASCOM-86 - MicroSoft	8085	395	279
CB-86 - DRI	CPM86	600	439
Prof. BASIC Compiler	PCDOS	345	325
BASIC Dev't System	PCDOS	79	72

EDITORS Programming

C Screen with source	8080/86	NA	60
EDIX - clean	PCDOS	195	149
FINAL WORD - for manuals	8080/86	300	215
MINCE - like EMACS	CPM, PCDOS	175	149
PMATE - powerful	CPM	195	175
	8086	225	195
VEDIT - full, liked	CPM, PCDOS	150	119
	8086	200	159

FORTRAN

MS FORTRAN-86 - Meg	MSDOS	$350	$255
SS FORTRAN - 86	CPM-86	425	345
FORTRAN-80 - 66 decent	CPM-80	500	350
INTEL FORTRAN - 86	IBM PC	NA	1400
DR FORTRAN COMING			
RM FORTRAN COMING			

LANGUAGE LIBRARIES

C to dBASE interface	8080/85	$125	$115
C Tools 1 - String, Screen	PCDOS	NA	115
C Tools 2 - OS Interface	PCDOS	NA	92
FLOAT 87 - Lattice, PL1	PCDOS	NA	115
GRAPHICS: GSX - 80	CPM80	NA	75
HALO - fast, full	PCDOS	200	175
Greenleaf for C - full, 200+	PCDOS	NA	165
ISAM: Access Manager - 86	8086	400	300
BTRIEVE - many languages	PCDOS	245	215
PHACT - with C	PCDOS	NA	250
FABS	CPM80	150	135
PASCAL TOOLS - Blaise	PCDOS	NA	115
SCREEN: Display Mgr. 86	8086	500	375
PANEL-86 - many languages	PCDOS	295	245
WINDOWS for C	PCDOS	NA	115
Virtual Screen - Amber	PCDOS	295	call

PASCAL

	ENVIRONMENT	LIST PRICE	OUR PRICE
PASCAL MT + 86	CPM86/IBM	$400	$289
without SPP	CPM80	350	249
MS PASCAL 86	MSDOS	350	255
SBB PASCAL-great, fast	PCDOS	350	315
PASCAL 64 - nearly full	COM 64	99	89
SBB Jr - best to learn	PCDOS	NA	95

OTHER PRODUCTS

Assembler & Tools - DRI	8086	200	159
COBOL - Level II	8086	1600	1275
CODESMITH-86 - debug	PCDOS	149	139
GC LISP	PCDOS	495	475
IQ LISP - full 1000K RAM	PCDOS	175	call
Janus ADA - solid value	PCDOS	500	449
MBP Cobol-86 - fast	8086	750	695
Microshell improve CPM	8080	150	125
Microsoft MASM-86	MSDOS	100	85
PL/1-86	8086	750	560
PLINK-86 - overlays	8086	350	315
POWER - recover files	8080/86	169	139
READ CPM86 from PCDOS	PCDOS	NA	55
READ PCDOS on an IBM PC	CPM86	NA	55
Trace 86	PCDOS	125	115

FEATURES

C INTERPRETERS for MSDOS - Ask about one for beginners for $85 or full development for $500.

C HELPER includes source in C for MSDOS or CPM80 for a DIFF, GREP, Flowcharter, C Beautifier and others. Manage your source code easier. $125.

PROLOG86 Interpreter for MSDOS includes tutorials, reference and good examples. Learn in first few hours. For Prototyping, Natural Language or AI. $125.

Call for a catalog, literature, and answers

800-421-8006

THE PROGRAMMER'S SHOP™

128-J Rockland Street, Hanover, MA 02339

Visa 617-826-7531, Mass: 800-442-8070 MasterCard

Note: All prices subject to change without notice. Mention this ad. Some prices are specials.

Ask about COD and POs.
Special formats available.

PROLOG-86™

Learn Fast, Experiment, Prototype

1 or 2 pages of PROLOG would require 10 or 15 pages in "C" - and PROLOG is much easier.

In **one evening** develop a feel for PROLOG. In a **few days** understand and enhance artificial intelligence programs included like:

*** an Expert System**
*** a Natural Language Processor**

PROLOG-86 includes two tutorials, a reference, 6 sample programs and a PROLOG interpreter.

Intro price: $125 for PCDOS, MSDOS or CPM-86. Call:

Full Refund if not satisfied in first 3 weeks.

SOLUTION SYSTEMS

45-J Accord Park Drive, Norwell, MA 02061

617-871-5435

C Helper™

UNIX-like Utilities for C Programming with source

Save time when working with your C programs. Full source lets you make them work your way, helps you learn.

Utilities included: compare files (DIFF), cross reference your variables (CCREF), examine the flow of functions as they call each other (FCHART), format and indent programs (pretty printer), search for patterns (GREP). Others check program syntax, print programs your way and more.

$135 for PCDOS, MSDOS, CPM-86 or CPM80.

Call with questions or for "Programming with C Helper".

SOLUTION SYSTEMS

45-J Accord Park Drive, Norwell, MA 02061

617-871-5435

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

PC MUSICIAN

Creating a Music Composition and Editing System

by Christopher Wiley

The development of PC Musician began with a purely personal need: I sing with a chorus and needed to learn my parts. Since I don't own a piano, nor can I play one very well. I naturally turned to my PC. A fairly complete search revealed no commercially available music editors for the PC, so I decided to write my own.

My original goal was simply to produce a system that would allow me to easily "copy" music from printed scores into the computer, which would then store the music to disk and play it on request. This would let me hear my choral part as often as I needed to (finally) get it straight.

At some point during the early work, it occurred to me that others might find such a program useful, and I began to think more carefully about the design of such a system.

DESIGN CONSIDERATIONS

The major technique I used to help formulate design goals for my music editor was to carefully visualize the typical user of the system. I imagined someone like myself, without the money to buy special synthesizer boards or the musical sophistication to require more than the PC's built-in speaker can provide.

This led to the primary goal that the system require no extra hardware purchases. As a result, a number of constraints became apparent: First, the program could not be icon-based and driven by a mouse, joystick, or other pointing device. Second, the music produced would be limited to one voice and constant volume. Next, the graphic display must accommodate both the monochrome and color/graphics adapters. And, finally, the memory requirements would have to be minimized.

It was also clear that my imagined user would not want to invest a great deal of time in learning the system. It must be easy to learn and use. The PC's BASIC interpreter already contains a fairly comprehensive "language" for producing music, but it is cumbersome, error-prone, and bears no resemblance to sheet music. My system must allow someone to copy sheet music without knowing which note is an "A" and which a "C".

To the greatest extent possible, the complexities of even simple sheet music must be shielded from the user. Thus, such things as time signatures, key signatures, accidentals, octaves, and measures must be handled as automatically and simply as possible. The user should be able to merely "copy" exactly what he sees on sheet music.

These considerations led to the decision that the system should be graphically based (with versions for both monochrome and color/graphics displays), primarily command-driven using the PC's cursor-control and function keys, intuitive and automatic where possible, and endowed with an on-line help system of some sort.

IMPLEMENTATION

In deciding how to meet these design goals best, my first problem was choosing the programming language. BASIC seemed an obvious first choice because of its extensive graphics and music commands, but I don't own the compiler, and I doubted that any interpreted program could be fast enough. Furthermore, I knew the code produced would be bulky and relatively unstructured.

My alternative was C, a language I was just learning. It clearly would be fast and compact enough, and the code would be

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

fairly structured. Unfortunately, my compiler provided no library functions for driving the speaker or manipulating shapes on the graphic display. I would need to create these.

I chose C, because solving the problems it posed would be a challenge and would extend my grasp of the language. So, the first task was to produce sounds on the speaker. By adapting assembly language routines that had been published in various magazines, I came up with a sound ()function which is called with two parameters: sound (freq. duration), where freq and duration are integers representing Hertz and milliseconds, respectively.

Now that I could run the speaker, I had to consider carefully how to tie the position of a graphic symbol on the screen with the pitch of a note, as is the case in sheet music. One alternative would be to store the screen symbol and position information with each note. However, in order to keep memory use down, I chose a different approach.

While note entry/editing is occurring, the cursor keys are used to move up and down on a blank music staff represented on the screen. The position of the cursor, as measured in screen rows 0 - 24, provides an index into an array containing the frequencies of notes for the selected staff and octave. Modifiers such as sharp or flat may be specified to alter the index and hence the pitch. Then a duration is chosen which determines both the duration of the note and the symbol used to represent the note on the screen.

Once the user is satisfied with the note, he then "enters" it by pressing the large "+" key on the numeric keypad. All the information about the note is now condensed into the two numbers required by the sound () routine, namely frequency and duration. These are stored in simple sequential array, both in memory and on disk.

This approach is simple and memory-efficient, but a price is paid. All the on-screen information about the note's symbol and position is lost once the screen is refreshed. Therefore, direct visual editing of the music can only occur as long as the notes are still on the screen. This also means that PC Musician is unable to show the notes "rolling by" on the screen as the song is played. That admittedly desirable feature would require either a lot of memory or extensive back-calculation or reconstruction of the visual appearance of the music.
Another problem raised by this visual approach to music entry is that various symbols are needed to represent different notes. Since I was using the screen's "rows" as the basis for my index into the frequency array, it seemed sensible to use character shapes for the notes. However, because I didn't own any sort of character or font editor, I was forced to write my own.

Once the font editor was working, I was able to create shapes for the notes, rests, and other musical symbols and place them in the upper 128 positions of the ASCII code. They could then easily be placed on the screen by PC Musician. An additional benefit of this scheme was the easy substitution of available built-in character symbols for the monochrome version of the program.

USER INTERFACE

As with any program, the user interface for PC Musician is of primary importance. An icon-based system held obvious appeal, especially for the musically naive user who could just "point" to the note he wanted and "drag" it over to the correct position on the staff. However, such an approach definitely requires a mouse or other pointing device. Most PC owners don't yet have one, and I didn't want to force them to.

A side benefit of the decision not to use icons was that the screen display could be kept simple and uncluttered. I think a major failing of many programs on the market today is excessive clutter on the screens. To avoid this, I also rejected the idea of menus during the creation/editing phase. Thus, a command- driven system emerged.

The problem with command-driven systems is that, until the user becomes expert, he may be unaware of the options available to him or be overwhelmed by their number. Commands must be organized so as to guide the user along a logical path. In PC Musician does this by taking advantage of the special keys on the PC keyboard and assigning them functions in a logical manner.

The user is confronted with a blank musical staff with a cursor sitting at the lower left corner. His first task (and impulse) will be to move that cursor to the appropriate spot. He does this with the cursor-arrow keys at the right end of the keyboard. Using the up-arrow, down-arrow, PgUp, and PgDn keys he can quickly position the cursor on the staff.

Next, he must select the duration or "type" of note. For this purpose the leftmost column of function keys were assigned note values in descending order: F1 = whole note, F3 = half note, andso on to shifted-F9 = 32nd note. When any of these keys is pressed, a visual confirmation appears in the upper left of the screen showing both the name and symbol for the note.

Any further modifiers required are assigned to the next column of function keys: F2 = dotted, F4 = sharp, F6 = natural, F8 = flat, and F10 = rest. More than one modifier may be required, so they may be combined. Again, reminder notices appear, and remain, in a window in the upper left of the screen, so by the time the user is ready to "enter" the note, a complete description of what he has specified is displayed on screen. If satisfied, he then presses the large

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

"+" key on the numeric keypad, and the note is recorded and played. The symbol appears at the cursor location, and the cursor advances one column to the right.

The user may keep one hand on the cursor keys and one on the function keys, and by proceeding in a logical sequence, reinforced by the location of the command keys, he can quickly and easily specify the proper note. Other commands are available by pressing single letter keys, mnemonically selected, on the main keyboard. These usually result in some "dialogue" with the user, which appears in the lower part of the screen.

If at any point the user is unsure of how to proceed, he merely follows the advice constantly displayed at the bottom center of the screen: Pressing a ? produces a help screen containing a brief description of every command, organized in a way to facilitate their use. Pressing any key will instantly restore the editing screen to its previous state. These switches are rapidly accomplished by swapping memory images of the two screens, using another assembly language function (see program listing).

I encountered a number of other design problems, such as how to handle "scrolling" when the user reaches the right end of the staff; how to implement true editing or changing notes, on both a note-by-note and block basis; and how to let the user access the disk directory to play some songs. These I'll leave aside in the hope that I've conveyed my approach to some of the major design and implementation issues involved in this project.

MARKETING

Now that I had a system consciously designed for a certain audience, how could I best reach them? Actually, the user-supported distribution scheme had been one of my early choices and had influenced my basic design criteria. It was important to me that PC Musician be as cheap as possible. Furthermore, I wanted to assure wide distribution so that most people could experiment with making music on their PC's. The user-supported scheme allows this with no risk to the user and a minimal financial burden for me.

Certainly I hope that PC Musician will be judged useful and enjoyable enough to deserve a contribution. There are definitely many improvements that could be made if I get the sense that people are using and enjoying the program.

Here is a section of code used to implement the help function:

Introducing the Creative Genius. . .

YOU. **Discover how easy programming can be with DataBurst.™** A unique runtime screen processor and source program generator, DataBurst™ will decrease your program development time and increase the value of your application programs. The *unique* DataBurst™ screen editor provides fast, easy screen design. Program independent screen formats reduce both development and maintenance time.

During execution of your program, DataBurst™ controls all user interaction through one assembly language interrupt service routine, requiring as little as 14K of memory. A true full-screen processor, DataBurst™ allows unlimited design complexity, and brings a mainframe advantage to your IBM® PC.

DataBurst™ is available through your local computer retailer or directly from Key Solutions, Inc. To order directly, please send check or money order for $225* to Key Solutions, Inc., P.O. Box 2297, Santa Clara, CA 95055. Additional language support (BASIC Compiler and C Compiler) is available for $40.* (Please inquire about release dates for other language interfaces).

IBM® is a registered trademark of IBM Corporation.
DataBurst™ is a trademark of Key Solutions, Inc.

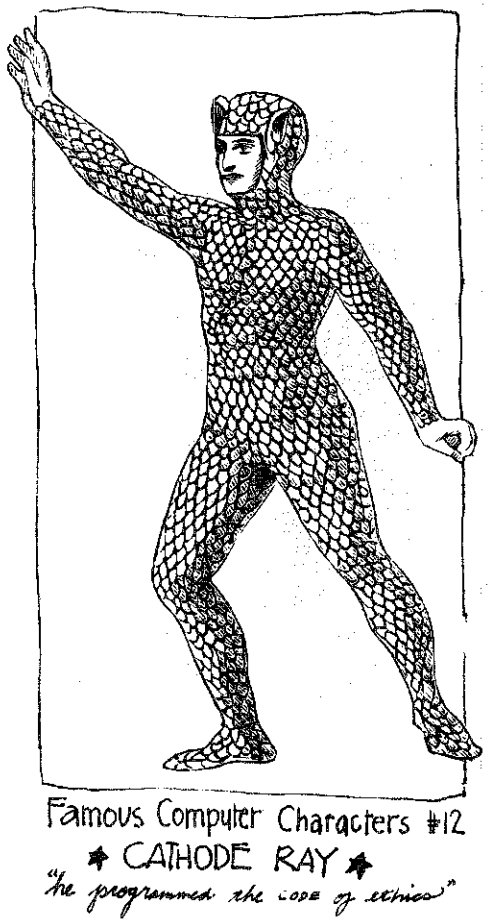
*In California add applicable sales tax.

© Copyright 1984 Key Solutions, Inc.

The Design Tool for the Creative Programmer

DATA BURST™

KEY SOLUTIONS™



PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
#define SCRNBUF 16000          /* these appear at the top of  */
#define STORE 1                 /* the source code file...     */
#define RESTORE 2              /*                             */

edit_screen=malloc(SCRNBUF);    /* this code is at the start of the   */
help_screen=malloc(SCRNBUF);    /* initialization, it makes room for  */
help_flag=0;            /* the screen images.                        */


help()        /* entered from case '?': statement in edit loop.       */
{
    char ch;

    if(help_flag){        /* means help screen has been built already   */
        swapper(STORE,edit_screen);        /* so store edit screen       */
        swapper(RESTORE,help_screen);    /* and show help screen       */
        locate(23,45);                    /* cursor to r23, c45         */
        ch=ci();                         /* await keypress...and       */
        swapper(RESTORE,edit_screen);    /* show edit screen.          */
    }
    else{                /* 1st time thru, must build help screen first  */
        swapper(STORE,edit_screen);        /* save current screen        */
        make_help();                  /* function to 'write' help screen   */
        swapper(STORE,help_screen);        /* now store it               */
        help_flag=1;                     /* and tell everyone          */
        ch=ci();                         /* await keypress...and       */
        swapper(RESTORE,edit_screen);    /* restore edit screen.       */
    }
}

; SWAPPER_ to store/restore any graphics screen image
;
;  usage   swapper(option, buffer pointer);
;            where option: 1=store screen,  2=restore screen
;
    cseg
    public swapper_    ;declare function as public for compiler

swapper_:
    push bp
    mov bp,sp         ;set up stack pointer to get parameters
    mov dx,[bp+4]     ;get option
    cmp dx,1         ;store image?
    je store         ;yes...go to it
    cmp dx,2         ;no. restore image?
    je restore         ;yes...go
    jmp return         ;no...bail out
store:
    push ds             ;save data seg
    mov di,[bp-6]     ;get buffer address
    mov ax,0b800h     ;load start of graphics screen
    mov ds,ax         ;and make it new data segment
    mov si,0         ;set index to beginning
    mov cx,8000         ;load word count
    cld                 ;clear direction flag
    rep movsw         ;move graphics screen to buffer
    pop ds             ;clean up
    jmp return         ;and quit
restore:
    mov si,[bp+6]     ;get buffer address
    push es             ;save extra seg
    mov ax,0b800h     ;graphics screen becomes
    mov es,ax         ;new extra segment
    mov di,0         ;index to start
    mov cx,8000         ;word count for full screen
    cld                 ;clear direction flag
    rep movsw         ;move buffer to graphics screen
    pop es             ;restore old es
return:
    pop bp             ;final clean up
    ret                 ;and return to C caller
    end
```

□

DeSmet C

The fastest 8088 C Compiler available

FULL DEVELOPMENT PACKAGE
- C Compiler
- Assembler
- Linker and Librarian
- Full-Screen Editor
- Newsletter for bugs/updates

SYMBOLIC DEBUGGER
- Monitor and change variables by name using C expressions
- Multi-Screen support for debugging PC graphics and interactive systems
- Optionally display C source during execution
- Breakpoint by Function and Line #

COMPLETE IMPLEMENTATION
- **Both** 1.0 **and** 2.0 DOS support
- Everything in K&R (incl. STDIO)
- Intel assembler mnemonics
- **Both** 8087 **and** Software Floating Point

OUTSTANDING PERFORMANCE

Sieve Benchmark

COMPILE	4 Sec. RAM — 22 Sec FDISK
LINK	6 Sec. RAM — 34 Sec. FDISK
RUN	12 Sec.
SIZE	8192 bytes

DeSmet C Development Package $159

To Order Specify:

Machine _____

OS ☐ MS-DOS ☐ CP/M-86

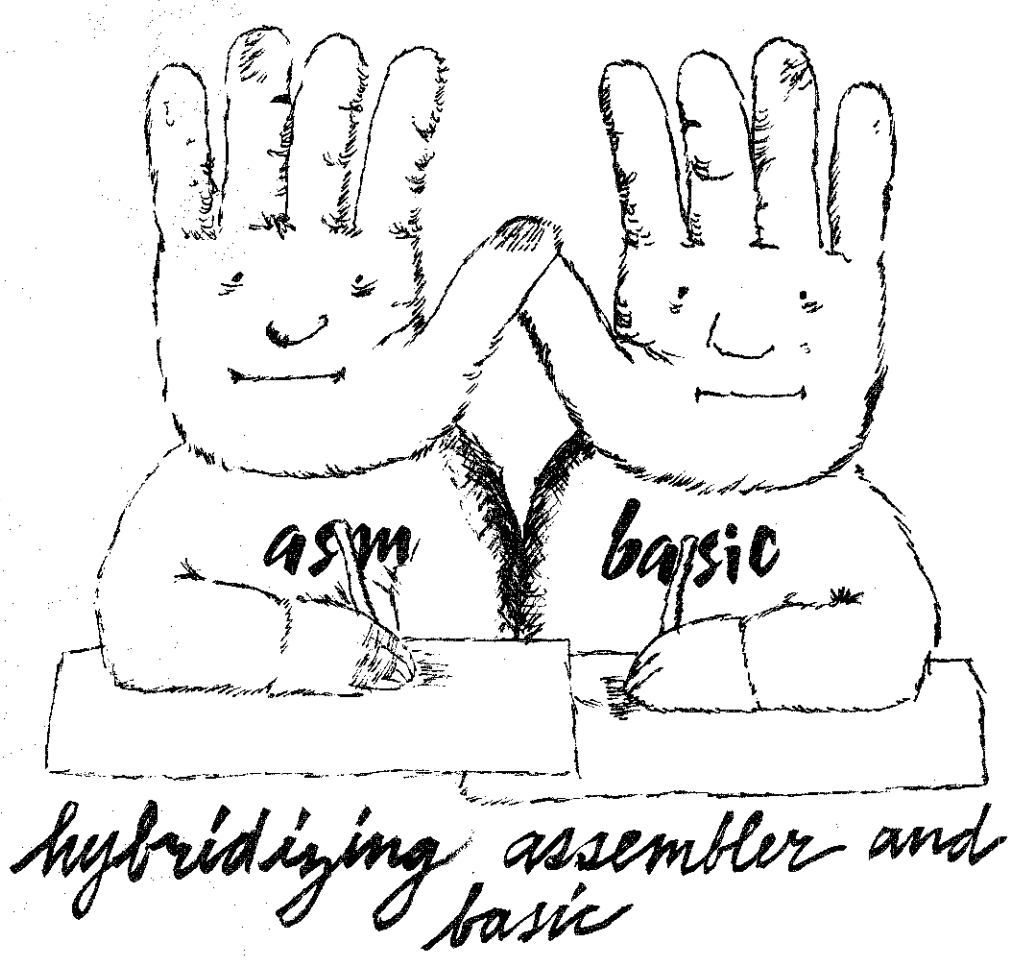
Disk ☐ 8" ☐ 5¼ SS ☐ 5¼ DS

CWARE CORPORATION

P.O. BOX 710097
San Jose, CA 95171-0097
(408) 736-6905

California residents add sales tax. Shipping: U.S. no charge, Canada add $5, elsewhere add $15. Checks must be on a US Bank and in US Dollars.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



HYBRIDIZING ASSEMBLY AND BASIC COMPILER ROUTINES

By Wat Buchanon, President, The FreeSoft Company

Although many PC programmers are proficient in BASIC and other high-level languages, few are assembler experts. The BASIC compiler allows you to write programs in a familiar language, but, even though it overcomes the interpreter's slowness, it still leaves you confined within BASIC's limits.

Many programmers have found that the built-in routines of BASIC are quite satisfactory for applications. The lowly INPUT statement becomes quite elegant to those who have undertaken the task of coding such a procedure in assembly. Efficient programming means reinventing the wheel as infrequently as possible.

In some applications, you need speed beyond what compiled BASIC can provide, and nothing works better than good old assembler. However, the procedure for linking compiled BASIC and assembly modules into one package is at best murky, and for this reason few will venture into this type of task. In this article, we'll show you how to include assembly language routines in compiled BASIC code and give you a window into the PC's built-in BIOS routines in the process.

Before we continue, let me say that I strongly suggest that you get a copy of the Technical Reference Manual. The DOS BIOS (Basic Input/Output System) routines described in it can save you many hours of development time, and they are available for your use with hardly any effort. In fact, we are going to use one of the routines for disk input/output in our sample programs: the INT 13H function.

This function interfaces every program, including DOS, with the disk drive hardware. By specifying different functions, you can easily access many different disk I/O jobs, such as reading or writing sectors, or formatting tracks. All of these functions are described in the Technical Reference Manual, but we'll concentrate on just one of them: the Sector Read function.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

To use this function, certain 8088 registers must be set up before the routine is called. Here is the list:

```
REGISTER        CONTENTS
--------        --------
   AH             2  This specifies the Read Sector function.
   AL             1  Number of sectors to read.
   DL             ?  Drive number (0=drive A, 1=drive B, etc.)
   DH             ?  Side number (0 or 1, always 0 for single-sided
                     diskettes).
   CH             ?  Track number.
   CL             ?  Sector number.
   ES:BX          ?  Segment and memory address to read sector into.
```

Once the registers are set up, the INT 13H instruction is executed. It reads the sector. Certain values are also passed back to registers, depending on the success of the read. If the carry flag is not set, the read was successful; otherwise, the AH register contains the error code.

Now, all of this may seem rather confusing, but it can be accomplished with a rather simple assembly language routine that receives values passed from the compiled BASIC variables, sets up the appropriate registers, calls the BIOS function, and returns the error status and error code through a compiled BASIC variable for processing. One such routine looks like this:

```
CODE            SEGMENT         'CODE'          Do not include these
                PUBLIC          DISKIO          comments in your source
DISKIO          PROC            FAR             code!
                PUSH            BP
                MOV             BP,SP           Save SP and BP registers
                MOV             SI,[BP]+12      Position SI to first
                                                passed variable
                MOV             AX,[SI]         Put it in AX
                MOV             SI,[BP]+10      Position SI to second
                                                passed variable
                MOV             BX,[SI]         Put it in BX
                MOV             SI,[BP]-8       Position SI to third
                                                passed variable
                MOV             CX,[SI]         Put it in CX
                MOV             SI,[BP]-6       Position SI to fourth
                                                passed variable
                MOV             DX,[SI]         Put it in DX
                INT             13H             Call the BIOS function
                RCL             AL,1            Put the carry flag into AL
                MOV             SI,[BP]+12      (AH has error code), and
                MOV             [SI],AX         put AX into first passed
                                                variable.
                POP             BP              Restore BP and stack and
                RET             8               return to compiled program.
DISKIO          ENDP
CODE            ENDS
                END             DISKIO
```

As you can see, the BASIC variables are passed in an upside-down manner beginning at 6 bytes above the stack pointer register when the routine is called. Each of the integer variables we are going to pass takes up 2 bytes of memory and can contain values from 0 to 65,535. It will be necessary to use single (4 byte) or double (8 byte) precision variables to pass values beyond this range.

If you go ahead and assemble this using the Macro Assembler, you should use the filename DISKIO.OBJ for the resulting object code. Do not use the linker program to produce an .EXE file! We'll combine the .OBJ file with the following compiled BASIC source code. If you don't have the Macro Assembler, program listing #1 below will allow you to create the DISKIO.OBJ module from BASIC.

Now we need some BASIC code that will take care of the keyboard and display routines, as well as pass the values to the DISKIO module we just created. Here is program #2, DISKREAD.BAS:

```
 5 REM ** Program Listing #1,  DISKIO.OBJ
10 REM ** Don't forget to enter BASIC using the command: BASIC/S:95
20 REM ** This creates the assembler object module DISKIO.OBJ
25 REM **  for those of you who don't have the Macro Assembler.
30 FOR Z=1 TO 95:READ A:A$=A$+CHR$(A):T=T+A:NEXT Z
40 IF T<>6400 THEN PRINT "Whoops!  At least one of the numbers in
   the DATA statements":PRINT"has been typed in wrong.  Check your
   listing again!":END
50 INPUT "Drive for DISKIO.OBJ (don't enter a colon) ";DR$
60 OPEN "R",1,DR$+":"+"DISKIO.OBJ",93:FIELD 1,95 AS K$:LSET K$=A$
65 PUT 1,1:CLOSE
70 PRINT:PRINT"Program DISKIO.OBJ has been created!":END
80 DATA 128,3,0,1,65,59,150,7,0,0,4,67,79,68,69,68,152,7,0,96
90 DATA 36,0,2,2,1,216,160,40,0,1,0,0,85,139,236,139,118,12,139,4
00 DATA 139,118,10,139,28,139,118,8,139,12,139,118,6,139,20,205
10 DATA 19,208,208,139,118,12,137,4,93,202,8,0,142,144,13,0,0,1
20 DATA 6,68,73,83,75,73,79,0,0,0,153,138,7,0,193,0,1,1,0,0,172
10 REM ** Program Listing #2, Accessing a raw disk sector through a
   BIOS call
20 REM ** from BASIC compiler source code.
25 DEFINT A-Z:DIM BUFFER1%(512)
30 INPUT "Lowest byte value to print (0 to 254) ";LB:IF LB<0 OR
   LB>255 GOTO 30
35 PRINT "Highest byte value to print (";LB+1;" to 255) ";:INPUT HB
37 IF HB<LB+1 OR HB>255 GOTO 35
40 INPUT "Drive (A=0, B=1, C=3, D=4) ";DR
50 IF DR<0 OR DR>4 GOTO 40
60 INPUT "Side (0 or 1) ";H
70 IF H<>0 AND H<>1 GOTO 60
80 INPUT "Track (0 to 39) ";T
90 IF T<0 OR T>39 GOTO 80
00 INPUT "Sector (1 to 9) ";S
10 IF S<1 OR S>9 GOTO 100
20 REM ** Lines 40 to 100 get values needed by BIOS call.  You can
   allow track
30 REM ** reads higher than track 39 by changing line 90.  Similarly,
   the
40 REM ** range of sectors allowed in line 110 is purely arbitrary.
50 A=0:B=0:C=0:D=0:CALL DISKIO(A,B,C,D)
60 REM ** Line 150 just resets the drives and prepares for BIOS calls.
70 A=2*256+1:REM ** This is the sector read instruction
80 B=VARPTR(BUFFER1%(0)):REM ** This is where the sector will go into
   memory.
90 C=T*256+S:REM ** These are the track and sector values to be passed
00 D=H*256+DR:REM ** These are the side and drive values to be passed
10 CALL DISKIO(A,B,C,D):REM ** Call the assembly routine passing
   values.
20 REM ** Now let's extract the value passed back from the assembly
   routine
30 REM ** in variable A that tells us whether our read attempt was
   successful
40 REM ** or not, and act accordingly.
50 AH=INT(A/256):AL=A-(AH*256):AH=ABS(AH):IF (AL AND 1)=0 GOTO 500
60 REM ** If read was successful, we branch directly to line #500.
70 REM ** Otherwise, we should print out what sort of error occurred.
80 IF AH=2 THEN A$="Address mark not found"
90 IF AH=3 THEN A$="Disk is write protected"
00 IF AH=4 THEN A$="Sector not found"
10 IF AH=8 THEN A$="Direct memory access overrun"
20 IF AH=9 THEN A$="Direct memory access crossed a 64K boundary"
30 IF AH=16 THEN A$="Cyclic redundancy check (CRC) error"
40 IF AH=32 THEN A$="Disk controller circuit failure"
50 IF AH=64 THEN A$="Track not found"
60 IF AH=128 THEN A$="Drive not ready. (Door open?  Disk mounted?)"
70 CLS:PRINT"ERROR!  The following happened during the disk read:"
80 PRINT"---> ";A$;" <---"
90 INPUT "Re-try read? (Y or N) ";A$:B$=LEFT$(A$,1)
100 IF B$="Y" OR B$="y" GOTO 150
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
110 IF B$="N" OR B$="n" GOTO 600
120 REM ** Execution branches to Line 500 after a successful read.
130 REM ** The actual sector data is held array BUFFER1%.  Since this is
140 REM ** an integer array (2 bytes for each element), it's easier to
150 REM ** do PEEKS using the VARPTR command than try to decode the individual
160 REM ** array elements into most significant and least significant bytes.
170 REM ** You can do whatever you like to the sector data from this point,
180 REM ** below we'll just print out the ASCII equivalent of the sector bytes.
200 K=VARPTR(BUFFER1%(0)):REM ** Find memory location of array.
210 CLS:FOR Z=K TO K+511:REM ** Print out 512 bytes (room in array for 1024).
220 K$=CHR$(PEEK(Z)):IF ASC(K$)<LB OR ASC(K$)>HB THEN PRINT "."; ELSE PRINT K$;
230 NEXT Z:REM ** Continue loop to end of sector
300 PRINT:INPUT "Read another sector? (Y or N)";A$:B$=LEFT$(A$,1)
310 IF A$="Y" OR A$="y" THEN RUN ELSE END
```

```
Type in the above program and save it in ASCII
(uncompressed) format using the BASIC command:
SAVE "DISKREAD.BAS",A
```

The whole trick in accessing an assembly module is to declare its name PUBLIC (as we did in the second line of the assembler code) and then execute it from the compiled BASIC program using the CALL command. If no values need to be passed to or from your routine, you can omit the parentheses and variables following the routine name. Otherwise, you'll need to study the assembler module and the BASIC program to see how we set up for variable passing. Notice how in lines 170 through 200 of the BASIC program we "stack" two singles byte values so they can be passed through one of BASIC's two-byte integer variables. This stacking process is reversed in line 250 when we want to check the error status of the operation. One more caveat: Always declare your assembly procedures FAR, so that the stack pointer used by the compiled program is left intact upon return from the assembly routine.

Once you've got the DISKIO.OBJ and DISKREAD.BAS programs on your disk, you are ready to compile and link them. (You did remember to save DISKREAD.BAS in ASCII format for the compiler, didn't you?)

The compilation phase is easy. Just tell the compiler the name of the BASIC program (in this case, DISKREAD.BAS). Once the program has been compiled, execute the LINK program. In response to the source files prompt, just type:

DISKREAD+DISKIO

and the linker should merge the two object files into a single .EXE file that can be executed. The order in which the object files are listed is important — the first file is regarded as the "parent" file that execution will begin with. If you have more than one assembler routine, just link them all together (the order of the routines isn't important) using additional plus signs and filenames, such as:

PROGRAM+ROUTINE1+ROUTINE2+ROUTINE3
+ROUTINE4 ....

The DISKREAD program itself needs a few additional comments: the low and high display values asked for in the beginning of the program allow you to set up a range of character values to be displayed. Values that fall outside of

PARALLEL PORTS

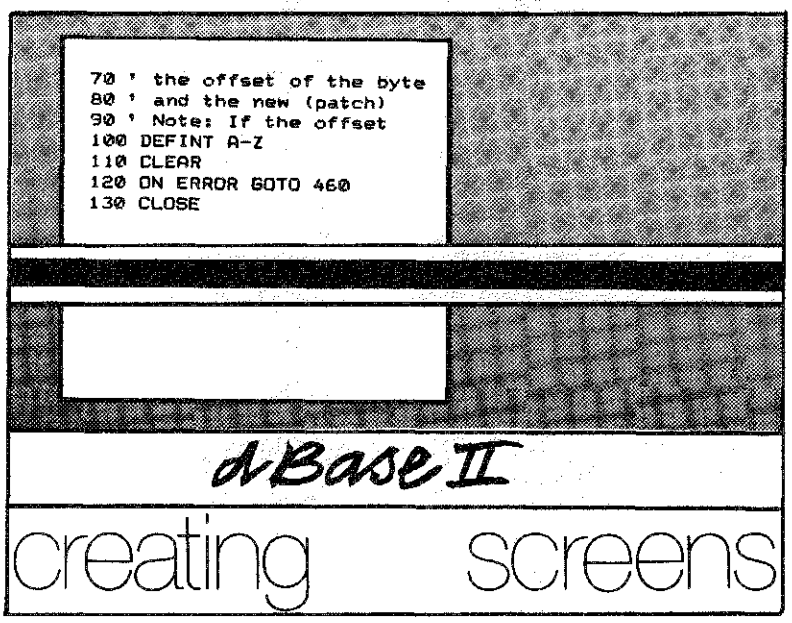
that range will be displayed as periods. If you want a nice clean display of alphanumeric characters, specify 32 as the low byte and 125 as the high byte. If you use the full character bandwidth of 0 through 255, you're likely to get some beeps and weird cursor control characters printed, which may make the display difficult to read.

The actual operation of DISKREAD is explained in REMark statements inside the program and should be studied carefully. I will, however, give you one last neat trick to play around with. A routine similar to lines 150 through 410 can be used to WRITE the memory buffer contained in the BUFFER1% array to a disk sector: simply change the A=2*256+1 statement in line 170 to A=3*256+1. I do advise extreme caution before playing around with raw disk sector writes, however, because indiscreet or accidental writes can easily destroy your disk files (or worse, the entire diskette) by wiping out the directory or file allocation tables. Always use a "scratch" diskette for experimentation.

By using hybrid compiler/assembly code, you can easily produce highly efficient programs in a relatively short time. Not only BIOS or DOS functions, but any assembler object module can be linked to compiled BASIC programs with this method.

□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



CREATING SCREENS

by Jim Bohle

Most of us would agree that developing applications in dBASE II is a relatively straightforward process. Several times I have simply defined a database and immediately started entering information. The more I read about dBASE II applications, the more I find this to be a typical scenario. Applications are created using dBASE II commands like CREATE, APPEND, EDIT, REPLACE. Then later, after the application is up and running, a number of programs are be written to formalize the process. I have often used the following command sequence to build databases:

SET TALK OFF
USE database
APPEND
USE
LIST FILES

The APPEND command, as well as others, automatically handle the screen layout, prompting, entry and editing of information. I have also found that a number of simple SET options allows me to customize the appearance and operation of data entry screens. I usually add some combination of the following SET commands:

SET TALK OFF	suppress display of results from commands
SET SCREEN ON	enable full-screen operations
SET COLON OFF	remove colons bordering data entry fields
SET BELL OFF	suppress bell on illegal data or field full
SET INTENSITY ON	display entry fields in reverse video
SET CARRY OFF	do not carry over data from previous record
SET CONFIRM OFF	skip to next field when maximum chars entered

Any of these SET options can be added to the above small program prior to the APPEND command. The option used is largely a matter of personal preference.

Part of the formalization process for applications involves adding command menus and enhancing the presentation of entry and display screens. There are numerous reasons for going beyond the simple, yet functional, display screens that dBASE II provides.For example, you may want to

* provide output as well as input fields

* present special field prompts, rather than structure names.

* exclude some fields from the data entry/edit process such as output-only or calculated fields.

* perform special field editing

* create a screen layout similar to a specific entry document

* customize the screen to improve operator convenience

* show special video attributes such as high intensity, underlining, reverse and blinking

Using most of these capabilities requires some extended dBASE II commands and a departure from the simple program example we started with. dBASE II presents several very useful tools for creating elaborate screens but this can involve a fairly significant amount of programming. To illustrate

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

some of these techniques, let me first define a simple database, APPLICNT.DBF, which might contain information on applicants for some type of activity:

01 appl:id	N, 5	application identification number
02 first:name	C, 12	applicant's first name
03 last:name	C, 15	applicant's last name
04 sex	C, 1	applicant's gender
05 birthdate	C, 8	applicant's birthdate
06 telephone	C, 12	applicant's telephone number
07 reg:fee	N, 6,2	application fee paid
08 age	N, 3	age of applicant
09 p:group	C, 1	participation group

Assume that the birth date and telephone fields are stored with standard editing delimiters and that the age and p:group fields are calculated (output-only) fields.

dBASE II provides the capability to place text and data any place on the screen with the "@ r,c SAY expression." For example:

```
@ 01,30 SAY "Application Database"
@ 03,60 SAY "Record: ' + #
@ 05,10 SAY "Application Id: "
@ 05,26 SAY appl:id
@ 18,40 SAY "Group: "
@ 18,47 SAY p:group
```

This sequence demonstrates how a particular record might be displayed according to a very specific screen layout. Note that row three displays the current record number using the '#' function. dBASE II version 2.4 (released July, 1983) also permits a relative address value for any of the row and column coordinates which allows more flexibility. You can reference a position as a relative change from the last "@ r,c SAY." For previous screen could be specified as:

```
@ 01,30 SAY "Application Database"
@ $+2,60 SAY "Record: " + #
@ $+2,10 SAY "Application Id: "
@ $,$+16 SAY appl:id
@ $+3,$+10 SAY "Group: "
@ $,$+47 SAY p:group
```

By defining an entire group of "@ r,c SAY" commands with relative addressing, you can flexibly position a sub-screen (or window-like display) wherever you desire under control of some input coordinates. We will talk about some very useful window techniques in a future article.

The "@ r,c SAY expression" command can be combined with a "GET variable" option to provide the means to accept information into a specified variable. As you can see from the example below, the GET option is coupled with the displayed prompt from the "@ r,c SAY" command. Accepting information for each GET variable is activated and controlled by a subsequent READ statement. When the full screen mode is in operation (SET SCREEN ON), the READ statement permits the cursor to be moved to any of the GET variables. For example, a simple menu for maintenance of the APPLICNT database could be specified as follows:

```
STORE ' ' to choice
DO WHILE choice
ERASE
@ 4,26 SAY "Applicant File Maintenance'
@ 5,26 SAY '- - - - - - - - - - - - - - - - - -'
@ 7,26 SAY 'A - add new applicants'
@ 8,26 SAY 'C - change applicants'
@ 9,26 SAY 'D - delete applicants'
@ 10,26 SAY 'L - list applicants'
@ 11,26 SAY 'E - exit'
@ 13,26 SAY 'Choice';
   GET choice PICTURE '!'
READ

** perform whatever processing is required - handle with
   CASE statement

ENDDO
```

The READ command requests an entry for "CHOICE." Note that the menu is repeated after each selection until you enter an 'E.' The new TEXT command provided in version 2.4 allows you to simplify screens by permitting a body of text, namely the above menu, to be easily routed to the display as follows:

```
TEXT
   Applicant File Maintenance
   ----------------------
   A - add new applicants
   C - change applicants
   D - delete applicants
   L - list applicants
   E - exit
ENDTEXT
```

As mentioned earlier,you often need specific editing control. The USING and PICTURE options provide a set of editing formats for the SAY and GET clauses, respectively. Several of these formats are used in the following ADD routine for our APPLICNT database:

```
STORE ' ' to idnum
@ 10,05 SAY 'Applicant Id: ';
        GET idnum PICTURE "XXXXX"
READ
IF idnum<>' '

** if applicant number is entered, see if it already exists
    FIND &idnum
    IF #=0

** applicant doesn't exist, so collect information and append record
        APPEND BLANK
        REPLACE appl:id with idnum
```


PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

```
        @ 12,05 SAY 'First Name: ';
                GET first:name picture "XXXXXXXXXXX"
        @ 12,35 SAY 'Last Name: ';
                GET last:name picture "XXXXXXXXXXXXXX"
        @ 14,05 SAY 'Gender: ';
                GET sex picture "!"
        @ 14,20 SAY 'Birthdate: ';
                GET birthdate picture "99/99/99"
        @ 16,05 SAY 'Telephone: ';
                GET telephone picture "999-999-9999"
        @ 18,05 SAY 'Regstr Fee: ';
                GET reg:fee picture "###.##"
        READ
** note - the two output fields 'age' and 'p:group' might be
** displayed on lines 14 and 16, respectively, after they
** are calculated following field entry (via READ)

        ELSE
** proposal number already exists, so give message

        ENDIF
ELSE
** no id number entered, so prepare to exit
ENDIF
```

A handy feature of the "@ r,c SAY" command is that you may save any screen (display, data entry, or menu), to a special .FMT file. The advantage here is the ability to modularize your programs and make use of the screen over and over again. To recall a screen definition stored in an .FMT file, use the 'SET FORMAT TO filename' command. The filename must have the extension .FMT, and contain only '@ .. SAY .. GET' lines and comments. The READ command initiates entry of the variables named in the GET subcommands. Again, Version 2.4 has made life easier.The "SET FORMAT TO" command can be issued and used with subsequent CREATE, APPEND, EDIT and INSERT commands,allowing you to create custom screens followed by simple APPEND and EDIT sequences.

```
SET TALK OFF
USE database
SET FORMAT TO APPLICNT
APPEND
USE
LIST FILES
```

The APPLICNT.FMT file contains a special display/entry screen for our applicant database.

Eventually, in the process of creating menu, display and data entry screens, one is intrigued by the possibility of using some of the special video attributes of a monochrome or color display. Some video features, like low and high intensity, underlining, reverse video (black characters on white background), and blinking serve to dress up a display. Version 2.4 dBASE II additionally has a "SET COLOR TO" option which controls colors on graphic monitors. However, the command also controls (albeit a bit mysteriously at times) the video attributes of the monochrome display. The command has the following format:

```
SET COLOR TO n1, n2
```

"n1" is the number used to select the desired color attribute for dim/reverse video display and "n2" is the desired color or terminal attribute for normal displays. After some frustrating experiments, (the documentation for this command is virtually nonexistent), I have discovered a set of control values for invoking most of the special monochrome video attributes.

```
n1, n2   video attribute
------- -------------------------
2, 113  low intensity, underline
2, 114  low intensity
2, 120  reverse video (black on white)
2, 121  high intensity, underline
2, 122  high intensity
2, 128  no echo, blank display
2, 129  blinking, low, underline
2, 130  blinking, low
2, 137  blinking, high, underline
2, 138  blinking, high
```

Other values will also produce some of these attributes — for example "2,2" will select normal intensity. This set is chosen since the values occur within a reasonably small range of numbers. The following command sequence demonstrates various attribute controls.

```
SET TALK OFF
STORE 'Add' to opt'name
ERASE


SET COLOR TO 2,121
@ 01,00 SAY "Applicant File Maintenance"
@ 01,60 SAY "Information Software"
SET COLOR TO 2,130
@ 02,00 SAY opt:name
SET COLOR TO 2,120
@ 02,72 SAY date()
SET COLOR TO 2,113
@ 06,00 SAY
SET COLOR TO 2,114
```

Experiment a little bit — there are many possibilities!

If you can write dBASE II programs and don't mind the labor involved, you can create an endless array of very elaborate and functional screens. With any programming language, you'll find that screen design and implementation can be a very time consuming endeavor. As you are undoubtedly aware, there are several products on the market which allow you to literally draw screens using an editor or even your favorite word processor. The utilities then proceed to analyze and generate dBASE II routines to implement the screen. Most of these permit you to specify not only the input and output fields, but special field attributes as well. In my next article, I will explore several of the most popular dBASE II program generators.

□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

LINKING ASSEMBLER SUBROUTINES TO PASCAL AND BASIC

by Corey Morrell

Recently, a friend asked me if there was any way that he could get the time of day from the system with greater precision than was given by BASIC's TIME$ statement (it returns time to the nearest second). I told him that he could access time to the nearest 1/100 of a second by using DOS and an assembler subroutine; one that issued a function call through interrupt 21 (see the DOS 1.1 manual, appendix D).

This example illustrates a problem that crops up periodically when you program in a high-level language like BASIC or Pascal: some things are impossible to do, messy to code, or very slow in executing. Frequently, the answer to this problem lies in writing an assembler subroutine to do the job and then calling that routine from a high-level language.

Many other magazines show examples of assembler subroutines, but they don't always describe how they can be connected to just any high-level language. Usually just one language interface is illustrated. So, let's explore how to call an assembler language subroutine from either interpretive BASIC or Pascal. I assume you have an assembler, or can cajole someone who does, into generating the code you need.

We'll illustrate the example I mentioned in the first paragraph — accessing the time of day from DOS. An identical assembler routine can be used for both BASIC and Pascal, the only difference being in the way the subroutine is interfaced.

Listing 1 shows an assembler subroutine that will return the time of day to the calling program. Note that the routine is defined as a PROC with the FAR and PUBLIC attributes. This is required in order to call it from another program. The routine works by issuing a function call 2C via interrupt 21. The function returns the time of day in four 8-bit registers as follows: hours in the CH reg, minutes in the CL reg, seconds in the DH reg, and hundredths of a second in the DL reg. Once extracted, the routine stores the numbers into the four variable addresses found on the Stack. Let's look at this again. When you need to pass information to or from an assembler routine, the IBM PC makes use of the Stack to pass that information. In your high-level language program you declare variable names that will transfer data. In our case, we need the information passed back. When a program issues a call to the subroutine, it creates a parameter list to be used by the assembler routine. The address of each parameter (variable) is pushed onto the Stack in the same order they appear in the calling statement. Once these are pushed, the system also pushes a 4-byte return address onto the Stack and then transfers control to the called routine. Keep in mind that the system Stack is built from high to low memory. The first parameter passed will end up at a higher address on the Stack than the second, and the 4-byte return address will be the lowest(or last) address. This is illustrated by an offset of +6 for the hundredths of a second, and offset +12 for the hours.

When the interrupt call has placed the time of day in the appropriate registers, the assembler routine must return values in the variables used by the program. The assembler routine knows where to store them because the variable addresses were stored on the Stack. So, MOV BX,[BP+06] gets the address of the variable that will hold hundredths of a second, and MOV [BX],DL stores the value in the DL register at that address.

Let's look quickly at the BASIC and Pascal programs that use this assembler subroutine.

THE PASCAL CONNECTION

Listing 2 shows a Pascal program that will call the assembler subroutine. The subroutine is a procedure and must be defined as external. The name of the procedure must be the same as the PROC name in the assembler subroutine. The word VAR at the beginning of the parameter list tells Pascal to pass the ADDRESS of each parameter rather than the number the variable may contain. This allows the subroutine to modify the variable in memory. If you leave the VAR out, then Pascal will pass the actual number the variable holds, and any changes made to it via the subroutine will be lost. So be sure and use VAR. Of course, if you don't want the variable's contents altered, then you can leave off the VAR.

When the subroutine returns to the Pascal program, the time of day will be in the variable names H, M, S, and HN. Our example just prints them out, but often you may want to save them for later use.

How do we connect these two pieces together? First, assemble the subroutine, giving the object code the name GETIME.OBJ. Then,compile the Pascal program, calling it TIMER.OBJ. Next — and here's the important part — run the Pascal linker. When it asks for the object files, type in: TIMER+GETIME. This tells the linker to create an .EXE file from TIMER (Pascal) and GETIME (assembler). To see the results, just type in the name you originally gave it.

THE BASIC CONNECTION

There are several ways to access assembler routines from the BASIC interpreter, but I will demonstrate only one of them here. This method works well for short subroutines like the one we're dealing with.

Listing 3 shows a BASIC program that uses our assembler routine. Since this is interpretive BASIC, with no compiling or linking, we must use a different interface method. Stated simply, we put the assembler's machine code into a BASIC

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

array, and then transfer program control to the address of the array. "Transfer control" means loading the instruction pointer (IP) with the array's address.

This is done as follows: dimension an integer array AR%(15) large enough to hold the assembler routine (Line 10). Each element in this array is two bytes long because of the % after the name. Lines 20-40 load the assembler routine into the array AR%. 'But where did you get the numbers?" you ask.

Compare the hexadecimal machine code on the left side of listing 1 with the DATA statements in lines 160 and 170 of listing 3. The machine code in listing 1 is produced by the assembler and is the hexadecimal equivalent of the executable machine language. If we can put that code, as is, into memory, by using an array, then we can run it by telling the program to execute instructions at the array's address.

There is one odd part about this arrangement. Every two bytes in the BASIC DATA statements are flipped around and are not in the same order as those in listing 1. Listing 1 says 55 8B EC B4 2C CD 21 8B 5E 06 etc., while listing 3 indicates 8B 55 B4 EC CD 2C 8B 21 06 5E. Why is this? When a 2-byte data item is stored (read) by a machine with an INTEL chip (8088 in an IBM PC), the low- and high-order bytes are reversed. Thus, if we want the bytes in our array to be 55 8B EC B4 etc., we must put them in our BASIC DATA statements as 8B 55 B4 EC etc. Then when they are read and flipped, they end up in the correct order.

Now that we have array AR% loaded with the machine code, all we need to do is get the address of the array and give control to the instructions at that address. Line 90 will get the address of the beginning of array AR% and store it in the variable name SUBR%. Line 100 is the actual CALL. Like our Pascal program, the addresses of the variables HRS%, MIN%, SEC%, and HUN% are pushed onto the Stack followed by the 4=byte return address. Control is then passed to the address stored in SUBR% which is just the beginning of our array. When it arrives there, the system finds 55 8B EC B4 2C etc., and starts executing the machine language instructions. On return to the BASIC program, the values we want are in the variables HRS%, MIN%, SEC%, and HUN%.

One last thing. The assembler program in listing 1 has a return instruction that says RET 8. The 8 tells the system how many bytes to POP off the Stack before it returns the (4=byte return address is not counted). Since we passed four 2-byte parameters (addresses), there are 8 bytes to POP from the Stack; hence the RET 8. Similarly, if we had passed only two parameters, the instruction would be RET 4. In general, the number in the return instruction should be two times the number of parameters you passed.

As I mentioned earlier, there are several ways to call assembler subroutines from BASIC. I'll mention two others briefly.

1) Instead of reading and loading the machine code into an array, you can load it somewhere else in memory outside of BASIC's 64K segment using the DEF SEG and POKE statements.

2) You can load the machine code directly from the diskette at execution time using the DEF SEG and BLOAD statements.

In summary, using assembler language subroutines together with your BASIC, Pascal, C or other high-level language programs will give you much more power and speed. As a small example, take a look at the functions that are available through INT 21 in appendix D of your DOS manual and use your right to assemble.

Listing 1

```
0000                            Stack   SEGMENT  PARA STACK  'STACK'
0000     40 [                           DB  64 DUP('STACK   ')
              53 54 41 43
              4B 20 20 20
                         ]

0200                            Stack   ENDS
0000                            CSEG    SEGMENT  'CODE'
                                        PUBLIC   GETIME
0000                            GETIME  PROC      FAR
                                        ASSUME  CS:CSEG,SS:STACK
0000  55                               PUSH BP
0001  8B EC                            MOV  BP,SP
0003  B4 2C                            MOV  AH,2CH
0005  CD 21                            INT  21H
0007  8B 5E 06                         MOV  BX,[BP+06]
000A  88 17                            MOV  [BX],DL
000C  8B 5E 08                         MOV  BX,[BP+08]
000F  88 37                            MOV  [BX],DH
0011  8B 5E 0A                         MOV  BX,[BP-10]
0014  88 0F                            MOV  [BX],CL
0016  8B 5E 0C                         MOV  BX,[BP-12]
0019  88 2F                            MOV  [BX],CH
001B  5D                               POP  BP
001C  CA 0008                          RET  8
001F                            GETIME  ENDP
001F                            CSEG    ENDS
                                        END
```

Listing 2

```
        PROGRAM TIMER(INPUT,OUTPUT);
  VAR H,M,S,HN:INTEGER;
(*                                                                    *)
PROCEDURE GETIME(VAR HRS,MIN,SEC,HUN:INTEGER);EXTERNAL;
(*                                                                    *)
BEGIN
    GETIME(H,M,S,HN);
    WRITELN('HOURS      = ',H);
    WRITELN('MINUTES    = ',M);
    WRITELN('SECONDS    = ',S);
    WRITELN('HUNDREDTHS= ',HN)
END.
```

Listing 3

```
10   DIM AR%(15)
15   '    LOAD ASSEMBLER ROUTINE INTO ARRAY  AR%
20   FOR N%=0 TO 15
30     READ AR%(N%)
40   NEXT
50   HRS%=0
60   MIN%=0
70   SEC%=0
80   HUN%=0
85   '    GET ADDRESS OF ARRAY AR% INTO SUBR%
90   SUBR%=VARPTR(AR%(0))
95   '    CALL ROUTINE TO GET THE TIME
100  CALL SUBR%(HRS%,MIN%,SEC%,HUN%)
110  PRINT USING "HOURS    = ##";HRS%
120  PRINT USING "MINUTES = ##";MIN%
130  PRINT USING "SECONDS = ##";SEC%
140  PRINT USING "HUNDRDS = ##";HUN%
150  END
155  '    HEXADECIMAL MACHINE CODE TO LOAD INTO ARRAY AR%
160  DATA &H8B55,&HB4EC,&HCD2C,&H8B21,&H065E,&H1788,&H5E8B,&H8808
170  DATA &H8B37,&H0A5E,&H0F88,&H5E8B,&H880C,&H5D2F,&H08CA,&H00
```

□

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

KERMIT

KERMIT COMMUNICATIONS PROGRAMS

The popularity of KERMIT has necessitated certain changes in their distribution policy. The following information outlines how you may receive the current release. For additional information contact Frank da Cruz, Manager of System Development, Univ. Center for Computing, Columbia University (NYC).

KERMIT is a protocol for transferring sequential files between computers of all sizes over ordinary asynchronous telecommunication lines using packets, checksums, and retransmission to promote data integrity. KERMIT is non-proprietary, thoroughly documented, well tested, and in wide use. The protocol and the original implementations were developed at Columbia University and have been shared with many other institutions, some of which have made significant contributions of their own.

KERMIT Implementations

KERMIT is presently available for the following systems, with others under development:

MACHINE	OPERATING SYSTEM	LANGUAGE
DECsystem-10	TOPS-10	MACRO-10
DEC-20	TOPS-20	MACRO-20
VAX-11	VMS	Bliss-32, Macro-32
IBM 370 Series	VM/CMS	IBM Assembler
CDC Cyber 170	NOS	Fortran-77
Honeywell	MULTICS	PL/I
VAX,PDP-11, SUN, etc.	UNIX	C
	UCSD p-System	UCSD Pascall, Macro-11
Terak, HP-98xx	RT-11	
PDP-11	CP/M-80	OMSI Pascal
8080,8085, or Z80	PC DOS, MS DOS	DR ASM
8086,8088		MS MASM-86
DEC Rainbow	CP/M-86	DR ASM86
Apple II 6502	Apple DOS	DEC-10/20 CROSS

The 8080 version runs on the DEC VT180 "Robin", DEC Rainbow-100, DECmate II (CP/M), Heath/Zenith-89 and 100, Intertec Superbrain, Apple II with Z80 Softcard, TRS-80 II (CP/M), Osborne, 1, Osborne Executive, Kaypro II, Vector Graphics, Ohio Scientific, Telcon Zorba, and others. The 8086 version runs on the IBM PC and lookalikes (such as the Compaq portable), the Heath/Zenith-100, and the Sirius/Victor 9000.

DISTRIBUTION POLICY

The KERMIT software is free and available to all. Columbia University, however, cannot afford to distribute free software on the scale required for KERMIT. Therefore, to defray our costs for media, printing, postage, labor, and computing resources, we must require a moderate donation from sites that request KERMIT directly from Columbia. The schedule is as follows:

Complete KERMIT Distribution $100.00
(Tape, Users Guide, and Protocol Manual)

Printed Documents $5.00 each
(Users Guide, Protocol Manual, or Any Source Listing)

The Users Guide contains complete instructions for installing and using each version of KERMIT. The Protocol Manual is a guide for writing a new implementation of KERMIT, and includes a source listing in the C language.

Other sites remain free to re-distribute KERMIT on their own terms, and are encouraged to do so, with the following stipulations: KERMIT should not be sold for profit; credit should be given where it is due; and new material should be sent back to Columbia University at the address below so that we can maintain a definitive and comprehensive set of KERMIT implementations for further distribution.

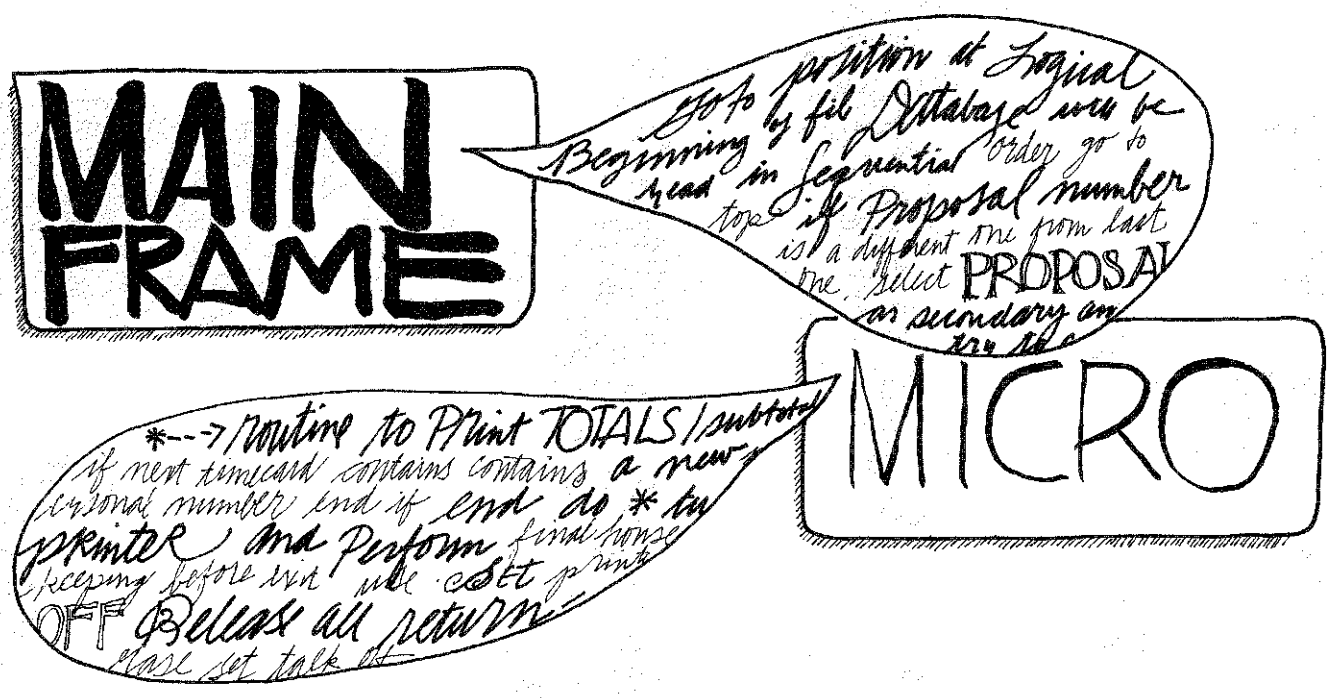
To order KERMIT from Columbia University, send a letter requesting either:

(a) The manuals or source listings you desire (specify each one), or

(b) A 9-track magnetic tape in one of the following formats:

SYSTEM	TAPE FORMAT	DENSITIES
TOPS-10	BACKUP/Interchange, Unlabeled	800, 1600
TOPS-20	DUMPER, Unlabeled	800, 1600
IBM VM/CMS	EBCDIC, CMS Format	1600 6250
	or EBCDIC, OS Standard Label	1600, 6250
OTHER	ASCII, ANSI Lable, Format 'D'	800, 1600

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED



(Specify system, format, and density.) One copy of each manual will be included with the tape. We will supply the tape, packaging, and postage.

We can only make tapes in the formats listed above. We cannot produce floppy disks; bootstrapping procedures are provided to allow the microcomputer versions to be downloaded from the mainframe for which the tape is produced. The tape includes all source programs, documentation, and, when practical, binaries or hex. Unfortunately, our limited resources do not allow us to provide automatic updates to KERMIT recipients when new implementations, documentation, or bug fixes appear.

Send your letter to:

KERMIT Distribution Columbia University Center for Computing Activities 7th Floor, Watson Laboratory 612 West 115th Street New York, NY 10025

Make check payable to: Columbia University Center for Computing Activities

Please list the machines and operating systems you expect to run KERMIT on, specify the tape format or the listings desired, and mention whether there are additional systems for which you require KERMIT or if you might be interested in attempting your own implementation for a new system.

KERMIT is also available to users of the BITNET network via a server at hos CUVMA. BITNET users may type "SMSG RSCS MSG CUVMA KERMSRV HELP" for further information. KERMIT is also available to users of ARPANET, via anonymous FTP from host COLUMBIA-20, in the area PS: KERMIT

No warranty of the software nor of the accuracy of the documentation surrounding it is expressed or implied, and neither the authors nor Columbia University acknowledge any liability resulting from program or documentation errors. □

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

MISC.

SOLID GOLD

There is a Programmer's Manual that is worth its weight in gold for anyone using DOS System Calls or other BIOS functions. It is even an excellent assembly tutorial book for using Macros. If you do any assembly language programming, then acquire a copy at all costs. If you have been wondering where some of that nice-looking ASM code has been coming from, this book is the place. Are you having trouble with the EXEC function, or setting up your registers correctly for a System Call? Wonder no more.

There are two ways to get the manual. If you are a member of Microsoft's ISV program you can call

Ken Sanders/ISV Program
Microsoft Corp.
10700 Northup Way
Bellevue, WA 98004
(206) 828-8080

and ask for a copy of the Programmer's Reference Manual for DOS 2.0 covering the System Calls. It is free via Microsoft. The other place to get a copy is to call Seattle Computer and ask for their MS-DOS 2.0 manual. The manual they sell has everything in it including the "Operating System, Programmer's Reference Manual." It includes a DEBUG, EDLIN, MASM, LIB, and USERS manual. You do not need the other sections, just ask them to break out the Programmer's Reference part. The whole manual is $75; the reference section can be priced separately.

Seattle Computer
1114 Industry Dr.
Seattle, WA 98188
(206) 575-1830

A similar version of this manual may be for sale at your local Radio Shack store; the new Tandy machine uses an Intel chip and MS-DOS. Check out the technical manual for their new 2000 computer.

MOZART AND BACH

Compose your own music, complete with bass and treble staffs. The PC Musician is a new member of the Freeware-program. With this package you can create, edit, store, and play back your New Wave rock songs. The requested donation is $20 if you enjoy the system. Send a formatted diskette and postage-paid return mailer to:

PC Musician
Christopher Wiley
Box 111, VAMC
Prescott, AZ 86313

COPY PROTECTION

Hoffman Computer Products, a leader in the mass diskette duplication business, has announced that they will also provide copy protection for those customers who desire it. Diskette duplication, protection schemes and packaging are available from one company.

Hoffman Computer Products
1720 Flower Ave.
Duarte, CA 91010
Call KCS&A, Sandy Black
(212) 682-6300

SCROLLMATE

This package adds additional scrolling space to your system. It saves any text that is displayed on your monitor and puts it in a buffer, up to 64K's worth. All programs that send sequential-style information to the screen will work with Scrollmate (DEBUG, DIR, TYPE, most compilers). The size of the save area is user-defined. A PHOTO feature is also included, allowing you to capture the screen buffer into a text file.

Scrollmate ($69.95)
Inner Loop Software
5456 McConnell Ave., Suite 120
Los Angeles, CA 90066
(213) 822-2800

SOFTWARE ESTIMATING

ATC is offering a software schedule and resource estimating tool. The program aids managers or consultants in determining staff levels, costs, milestones, and time factors. The user provides data from which the coding, development, and testing cycle are ascertained.

Software Resources ($175)
ATC
Rt. 2, Box 448
Estill Springs, TN 37330
(615) 967-9159

LEGALLY SPEAKING

Lawyers who use IBM PC's have their own newsletter, THE LAWYER'S PC. A full year's worth of back issues is already available. Subscriptions are $58; back copies are $3 apiece.

THE LAWYER'S PC
R. P. W. Publishing Corp.
P.O. Box 1108
Lexington, SC 29072

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

8086 AND 8088 ASSEMBLER CARD

The ultimate assembly-language instant-reference chart has been created by Micro Logic Corp. An 8 1/2 by 11 sheet is chock full of information, and is preserved, credit-card style for durability. The Micro Card shows the complete instruction set, Cycle Codes, Hex/Decimal conversion tables, Pin-outs, Second byte table, and miscellaneous notes. Order chart #7 at $5.95 a shot.

Micro Logic Corp.
P.O. Box 174
100 2nd St.
Hackensack, NJ 07602
(201) 342-6518

A REVIEW OF 1DIR

by Richard Wanderman

1dir is a "visual shell" that makes using the DOS operating system easier by presenting options and information graphically in a modular menu. You maneuver around in the menu with the cursor keys, pointing at the file(s) you want to handle, and indicating what you want to do with them.

The program is not a DOS tutorial — although it comes with helpful information on using DOS commands — it's a utility.

1dir is loaded into memory either by typing "1dir" or by including it in an autoexec.bat file. 1dir takes up about 15K of space on top of the DOS.

The 1dir screen is divided into modules.

USEFULNESS/APPLICATION

Besides helping with the syntax and execution of DOS commands, and displaying disk, memory and key toggle information at all times, 1dir makes dealing with long, complex hierarchical directories on a hard disk much easier. It displays "volume labels" in a reverse video and shows which sub-directory you're in. You can scroll through a directory in either direction and sort it by name, extension, size and date (although the dates themselves curiously don't show). You can also search through a directory for particular names or extensions or groups of them.

1dir comes with 30 small (2K) help files, each having a ".hlp" extension. There is a help file for each DOS 2.0 command (example: TREE.HLP). When you "type" these files, a short message describing the DOS command and its use appears on the screen.

The 1dir user's manual is relatively clear and concise. It comes with an excellent companion manual, "File Management Systems," that goes into detail about setting up files in the DOS 2.0 hierarchical directory system.

1dir has a built-in screen dimmer that turns off your screen in five minutes if you don't touch the keyboard (hitting any key turns it back on). You can change the time lag to anything between one and nine minutes or you can disable it completely. The screen can be turned off anytime by hitting Ctr1-0 from the main 1dir menu.

The program uses color which can be customizd.

You can change the default settings of 1dir so that it will boot up in a number of ways: with the command cursor on the command you use most often, with the directory sorted the way you want it, or with the type of system statistics you need displayed on the screen.

This is a well designed and useful program for people new to computers or to DOS. It's also useful for anyone with a hard disk desiring some help with complex directories.

1dir version 2.0 ($95)
Bourbaki, Inc.
431 Main St.
Boise, Idaho 83702
(208) 342-5849

INTERNATIONAL PC OWNERS

IPCO

"The international user group for the PC"

- **Software library of 175+ programs @ $3 each**
- **Datacount® General Ledger/Manual $35**
- **Volkswriter® $125**
- **Bimonthly newsletter 40+ pages**
- **Worldwide information exchange**
- **"Freeware"**

FOR FREE CATALOG WRITE:

INTERNATIONAL PC OWNERS, INC.
ATTN: FREE CATALOG
P.O. BOX 10426
PITTSBURGH, PA 15234

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

DEBUGGING SYSTEM

PC PROBE is a software and hardware debugging system from Atron. Quantity one runs in the $2,495 range; however, the product can be purchased unbundled. Separately, you can buy the manual, logic probes, power supply, PC Probe debugging software, and performance and timing analyzer. Features include: symbolic debugger, program crash recovery, user-defined macros, enhanced human interface, high-level language support, conditional constructs, external console option and 16 breakpoints. You get a diskette and external switch box that is compatible with any PC-type computer.

PC PROBE
Atron
20665 Fourth St.
Saratoga, CA 95070
(408) 741-5900

PASCAL MAGAZINE

Programmers who use IBM PC versions of Pascal have joined together and are producing a publication called DR. SAWBUCK'S SOFTWARE QUARTERLY. Subscriptions are $10 a year.

DSSQ
3-Ring Publishing Co.
Dept. BSF2
P.O. Box 2633
Dublin, CA 94568

ARTIFICIAL INTELLIGENCE

Here is a current list of magazines, journals and associations focusing on AI.

American Assoc. for Artificial Intelligence
445 Burgess Dr.
Menlo Park, CA 94025
(415) 328-3123

AI Magazine ($25)
AAAI
445 Burgess Dr.
Menlo Park, CA 94025
(415) 328-3123

AI Publications ($200 per year)
95 First St.
Los Altos, CA 94022
(415) 949-2324

AI Journal ($214.62)
Elsevier Science Publishers, Inc.
Journal Information Center
52 Vanderbilt Ave.
New York, NY 10017
(212) 867-9040

PC FIRING LINE

The programmer's DISK magazine, PC FIRING LINE, is continually extending its coverage. It regularly has articles on BASIC, C, Pascal, Assembly, Ada, C, Lisp, etc. The columns and source code are all there on diskette for your use. The most recent issue (#2) featured some beginning assembly listings, a copy of the Programmer's Calculator graphic maze program, the C source code for SQUEEZE and UNSQUEEZE (text compression programs), and articles on Forth, Ada, and Lisp.

Send $12 for a copy (two diskettes) to:

ABComputing
P.O. Box 5503
North Hollywood, CA 91616-5503
(818) 509-9002

SUPER ROM

A Canadian company has produced a Super BIOS chip, which you can use to replace the normal one IBM supplies. Once you install it you may bypass the memory check, change the beep sound, choose to continue or reboot after a parity check, redefine the color character set, and select upper/lower case startup. One additional feature is the ability to include a password, so that unauthorized users cannot even get through to DOS. The company also offers a complete updated and debugged BASIC ROM set of four chips. The BIOS ROM costs $40.

Digital Data
625 West Mall, Suite 1009
Toronto, Ontario
Canada M9C 4W9

UNIX PUBLICATION

A new publication, UNIX WORLD, presents numerous articles on using Unix in the office. A full-featured magazine, UNIX WORLD covers the application as well as the technical market. Volume 1 Number 2 has a review of PC/IX and a discussion of criteria for selection of an accounting system. One excerpt of note from this issue: Bob Freund, Director of Software Products for National Semiconductor, commented on a new computer benchmark by saying, "The port of Unix to the Cray-2 is reported to be so fast that a current benchmark indicates an infinite loop will complete in less than 60 seconds."

UNIX WORLD ($18)
Subscription Services
P.O. Box 1165
Dover, NJ 07801
(415) 949-3737

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

CP/M-86 AND PC-DOS

American CompuSoft has announced MatchPoint/86 for the IBM PC, a program that allows you to run CP/M-86 software under PC-DOS. You need DOS 2.0 to use it. The procedure consists of copying the CP/M-86 files onto a DOS formatted diskette using a supplied utility, and then loading MatchPoint-86 which will take care of the rest.

American CompuSoft
23113 Plaza Pointe Dr., Suite A
Laguna Hills, CA 92653
(714) 472-8186

UNIX 'MAKE' ON THE PC

Polytron Corp. has rewritten the Unix MAKE utility for the PC. Makefiles developed on Unix mainframes can run directly under MS-DOS using their POLYMAKE program. When a change is made to any module in a software system, POLYMAKE will automatically reconstruct the other components that are affected.

POLYMAKE ($99)
Polytron Corp.
P.O. Box 787 DS 1-399
Hillsboro, OR 97123
(800) 547-4000

SOFTWARE SEARCHING

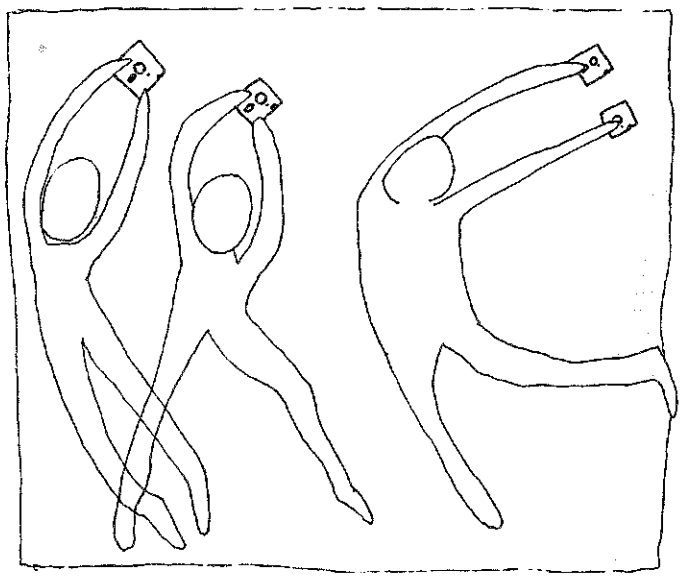
Finally, an on-line FREE database is being provided that will allow you to search for a particular software package you need and find out if it already exists. Searchmart Corp., in South Florida, is providing the service. Skip all the outdated PC Software Directory books that are three inches thick and outdated. It is truly an "On-Line Software Library," as advertised. Contact Victor Grundeau at (305) 845-2997 or Mary K. Hamm at (305) 845-2998 for the on-line number.

Searchmart Corp.
636 US Highway 1, Suite 210
North Palm Beach, FL 33408
(305) 845-2996

CABLES GALORE

Compu-Link has a wide variety of cables for the IBM PC at very competitive prices. The more you buy the higher the discount. Keyboard extension cables start out at $13 for one, or $11 for three or more. Power cable extensions are just $5. They also carry cables for your monitor, HP Plotter, parallel printer, and many more.

Compu-Link
P. O. Box 1604
New Port Richey, FL 34291
(813) 842-6628



SCREEN SAVER

Logical Systems Corp. has released a series of preventative maintenance utilities for the PC. They were developed for in-house use. The programs provided are: SCRNSAV, which blanks your screen after a defined period of time; DSKTIMER, that keeps track of the wear on your diskettes; MEMTST, a complete memory tester; DSKTST, a confidence test for drives and diskettes; and DSKCLN, a head-cleaning program. Very reasonably priced.

Screen Saver
Logical Systems Corp.
Route 1, Box 253
St. Michael, MN 55376
(612) 497-3861

IBM SEMINAR PROCEEDINGS

The Entry Systems Division of IBM puts on periodic workshops at computer faires. A summary of the topics they cover is compiled into a publication for software developers. There are at least three issues of the proceedings booklets out. For a copy write:

IBM, Personal Computer Seminar Proceedings
5Q9 / 237-3
P. O. Box 1328
Boca Raton, FL 33432

They discuss issues concerning compatibility, operating system particulars, 8087 notes, .EXE versus .COM files, and Macro Assembler specifics.

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

FREE SOFTWARE SUPPORT

"And he cast himself into the deep." It seems that Burton Bhavisyat has offered to answer any and all questions about CP/M, Pascal, Unix/Xenix, WordStar, and general programming techniques. Send a self-addressed, stamped envelope and he will respond to your needs. What more could you ask for?

Burton Bhavisyat
Rt. 1, Box 331
Moundsville, WY 25041

COMPUTER CONTROL

Sybex has published a book detailing how to hook up your IBM PC so it is in control of your office or home environment. THE IBM PC CONNECTION shows how a PC can take charge of a burglar alarm system, temperature, lights, and other devices. The author guides you through the simple steps of plugging in your household appliances. Analog to digital, and speech synthesis are covered too.

The IBM PC Connection ($16.95)
Sybex Press
2344 6th St.
Berkeley, CA 94710
(415) 848-8233

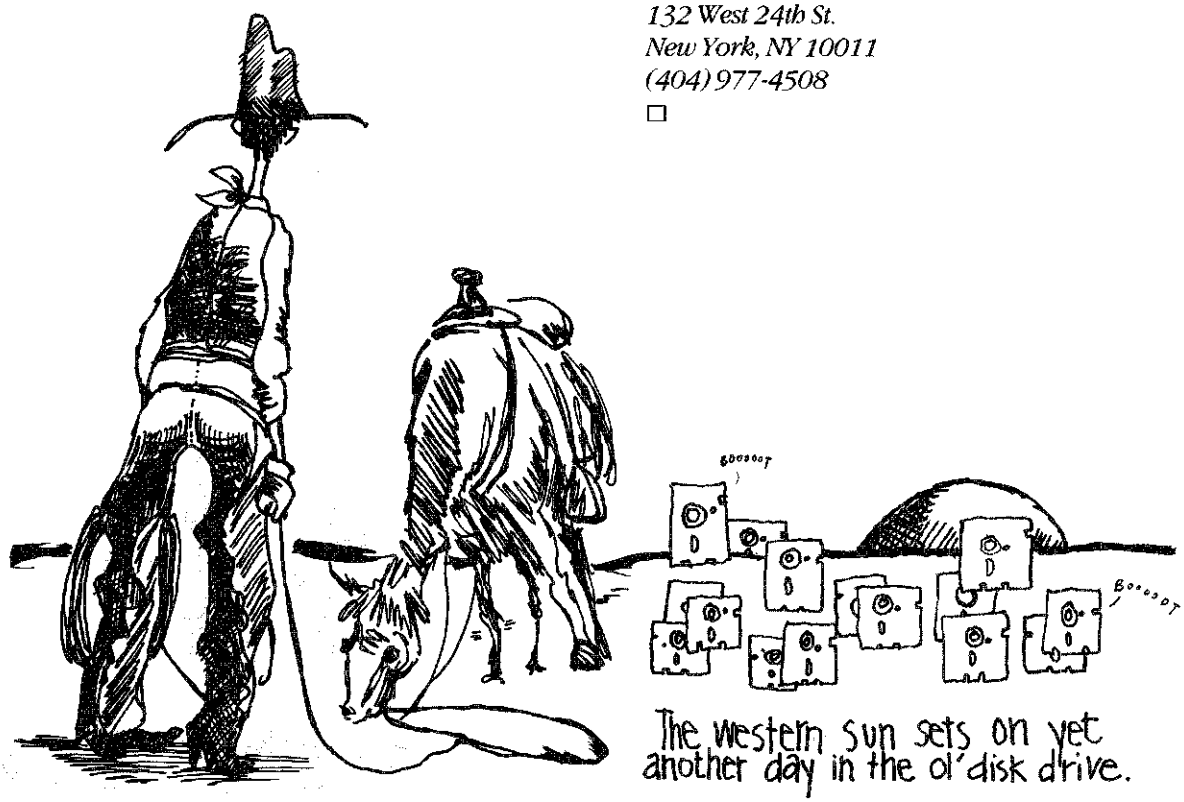
RANDOM NOTES

The SOURCE has connected itself to the E-COM mail service. E-COM allows you to send a message to a friend via your computer. You use your IBM PC word processor to type in the message, call up the SOURCE and send it to the E-COM mailbox. The SOURCE then takes the message, prints out a hard-copy version, and delivers it to the U.S. Postal Service. No secretary or stamp licking required.

MONOCHROME TO AMBER

Have you been looking longingly at the amber screens on the market, but can't get up the courage to shell out the bucks? You already have the IBM PC Monochrome, and few of the amber monitors even come close to its resolution, and the ones that do are in the $200-$300 price range. There is a solution. Langley-St. Clair sells an exact duplicate of the Monochrome *tube* for $100 with amber phosphor. You take out your green tube and install the amber one with the instructions they give you. Very simple. (Editor: I made this conversion myself, and it is quite simple.) For the $100, you get a lead impregnated tube (cuts down on emissions), an amber screen, and clearer text, because the background is blacker. It gives you *exactly* the same character set and resolution of the Monochrome display. I feel it is easier on the eyes. 30-day money-back guarantee. They also have the best set of articles about VDT problems. Just ask them for a copy.

IBM PC Amber Tube ($100)
Langley-St. Clair
132 West 24th St.
New York, NY 10011
(404) 977-4508
□



PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Programmer's Journal has a GREAT NEW SERVICE

Introducing a new feature! PROGRAMMER'S JOURNAL announces a clearing center of current opportunities for IBM PC programmers who have programs to sell, companies who are looking for IBM PC programmers, and those who may want to exchange ideas with other professionals in the field.

Simply put your message in 25 words or less and include your address.

Cost is a modest $50.00 per issue to reach the leading people and companies in the IBM PC programming world.

PROGRAMMERS' EXCHANGE

Converting Apple line to IBM PC. Need programmers. Contact:
Mere Inc.
Box 5465
Eugene, OR 97405

IBM PC programs wanted! Looking for state-of-the-art educational and personal productivity products. Contact:
PM INC.
Box 1526
Novato, CA 94947

Developing IBM authoring system for educational line. Write:
Universal Systems for Education, Inc.
14901 E. Hampden Avenue, Suite 250
Aurora, CO 80014

Leading Educational Developer needs system designer and programmers for major new product. Contact:
Avant-Garde Publishing Corp.
Box 30160
Eugene, OR 97403

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

ARE YOU AN ISSUE SHY OF A FULL SET?

Complete your library! Order any back issues you may have missed — or become a new subscriber. Use the form below or the handy subscription envelope provided. You'll be glad you did.

- **Vol. 1, No. 1:** Norton Pascal I/O, Program Style Manual, COBOL, System Analysis, BASIC modules, Rounding error.
- **Vol. 1, No. 2:** Defining character sets, C86 BIOS calls, Copy protection, Pascal code, Input routines, Writing an Assembler.
- **Vol. 1, No. 3:** ASM dumb terminal, C source code, ASM graphics line drawing, Text editors reviewed, Utilities.
- **Vol. 1, No. 4:** Diskette directory via Pascal, MASM WHILE-WEND, C/ASM interface, ASM stay resident routine, Structured programming.
- **Vol. 1, No. 5:** Multi-tasking software review, Transporting C Programs, Programmers calculator, dBase bargraphs, Utilities.
- **Vol. 1, No. 6:** ASM printer configuration, DOS patch utility, Hi-speed graphics, Software ownership, dWINDOW review.
- **Vol. 2, No. 1:** Freelance consulting, Sorts in C, Pascal text editor source code, Designing national database, Ultra utilities.

Name _____

Company _____

Address _____

City _____ State _____ Zip _____

Telephone _____

Send me the following back issues at $4.00 each:

_____ Vol. 1, No. 1
_____ Vol. 1, No. 2
_____ Vol. 1, No. 3
_____ Vol. 1, No. 4
_____ Vol. 1, No. 5
_____ Vol. 1, No. 6
_____ Vol. 2, No. 1
_____ Vol. 2, No. 2

_____ 1 year bi-monthly subscription
at $24.00 ($30 Canada, $40 other, U.S. funds only)

_____ Check enclosed _____ Bill me

_____ Charge my Bankcard: ___ VISA ___ MC ___ Am Ex

_____ Expir. Date _____

Signature _____

Send to:
PROGRAMMER'S JOURNAL
P.O. Box 30160
Eugene, OR 97403
(503) 345-3043

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED

Target Machines

Support Utilities

Fastest Assembler

Microsoft Network

Compilers of Choice

Multi-Tasking Development

Papa Bear, Mama Bear
& Baby Bear

PRODUCED 2004 BY UNZ.ORG
ELECTRONIC REPRODUCTION PROHIBITED